UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL

ADRIANO FRANCISCO RONSZCKA

CONTRIBUIÇÃO PARA A CONCEPÇÃO DE APLICAÇÕES NO PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON) SOB O VIÉS DE PADRÕES

DISSERTAÇÃO

CURITIBA
2012

ADRIANO FRANCISCO RONSZCKA

CONTRIBUIÇÃO PARA A CONCEPÇÃO DE APLICAÇÕES NO PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON) SOB O VIÉS DE PADRÕES

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de “Mestre em Ciências” – Área de Concentração: Engenharia de Computação.

Orientador: Prof. Dr. Jean Marcelo Simão
Co-orientador: Prof. Dr. Paulo Cézar Stadzisz

CURITIBA
2012

Dados Internacionais de Catalogação na Publicação

R774 Ronszcka, Adriano Francisco
Contribuição para a concepção de aplicações no paradigma orientado a notificações (PON) sob o viés de padrões / Adriano Francisco Ronszcka. – 2012.
236 p. : il. ; 30 cm

Orientador: Jean Marcelo Simão.
Coorientador: Paulo Cézar Stadzisz.
Dissertação (Mestrado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. Curitiba, 2012.
Bibliografia: p. 226-236.

1. Paradigma orientado a notificações. 2. Software aplicativo – Desenvolvimento. 3. Pórticos estruturais. 4. C++ (Linguagem de programação de computador). 5. Simulação (Computadores). 6. Engenharia elétrica – Dissertações. I. Simão, Jean Marcelo, orient. II. Stadzisz, Paulo Cézar, coorient. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. IV. Título.

CDD (22. ed.) 621.3

Biblioteca Central da UTFPR, Campus Curitiba

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Câmpus Curitiba

Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

Título da Dissertação Nº. 608

"Contribuição para a Concepção de Aplicações no Paradigma Orientado a Notificações (PON) sob o viés de Padrões"

por

Adriano Francisco Ronszcka

Esta dissertação foi apresentada como requisito parcial à obtenção do grau de MESTRE EM CIÊNCIAS – Área de Concentração: Engenharia de Computação, pelo Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial – CPGEI – da Universidade Tecnológica Federal do Paraná – UTFPR – Câmpus Curitiba, às 14h do dia 31 de agosto de 2012. O trabalho foi aprovado pela Banca Examinadora, composta pelos professores:

Prof. Jean Marcelo Simão, Dr.
(Presidente – UTFPR - CT)

Prof. Fabrício Enembreck, Dr.
(PUC-PR)

Prof. Cesar Augusto Tacla, Dr.
(UTFPR - CT)

Prof. Richardson Ribeiro, Dr.
(UTFPR-PB)

Visto da coordenação:

Prof. Ricardo Lüders, Dr.
(Coordenador do CPGEI)

AGRADECIMENTOS

Certamente estes parágrafos não irão atender a todas as pessoas que fizeram parte dessa importante fase de minha vida. Portanto, desde já peço desculpas àquelas que não estão presentes entre essas palavras, mas elas podem estar certas que fazem parte do meu pensamento e de minha gratidão.

Aos Professores Dr. Jean Marcelo Simão e Dr. Paulo Cézar Stadzisz por suas orientações e empenho para a realização deste trabalho.

À Universidade Tecnológica Federal de Paraná - UTFPR e ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial – CPGEI pela estrutura oferecida.

Aos amigos, Danillo e Glauber, gostaria de externar minha satisfação de poder conviver com eles durante a realização deste estudo.

Agradeço aos pesquisadores e professores da banca examinadora pela atenção e contribuição dedicadas a este estudo.

Gostaria de deixar registrado também, o meu reconhecimento à minha família, pois sem o apoio deles seria muito difícil vencer esse desafio.

E por fim, e nem por isso menos importante, agradeço a CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pelo apoio financeiro.

RESUMO

RONSZCKA, Adriano F. Contribuição para concepção de aplicações no Paradigma Orientado a Notificações (PON) sob o viés de padrões. 2012. 236 f. Dissertação (Mestrado em Engenharia de Computação) – Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial,Universidade Tecnológica Federal do Paraná. Curitiba, 2012.

A materialização original do Paradigma Orientado a Notificações (PON) implementada na linguagem de programação C++ possibilitou a criação de aplicações sob o domínio desse paradigma. Apesar de tais contribuições para com o paradigma, o desenvolvimento de aplicações no PON ainda apresenta baixo nível de maturidade e certo nível de dificuldade. Tais dificuldades advêm principalmente da nova forma de estruturar os programas, onde os mesmos seguem um fluxo de notificações, o que difere da programação convencional. Ademais, até o momento, apenas algumas aplicações foram desenvolvidas, com escopos relativamente pequenos, o que não contribui efetivamente para a consolidação do paradigma e, particularmente, para o aprendizado via um conjunto efetivo de exemplos. Neste sentido, primeiramente, a própria estrutura geral do *framework* foi relida e reapresentada sob o viés de padrões de projeto, assim como houve contribuições na própria implementação desse. Subsequentemente, este trabalho teve como objetivo principal de estudo o nortear do desenvolvimento de aplicações baseadas no PON de maneira mais simplificada e eficiente, usando os avanços aqui citados. Isso se deu particularmente pela apresentação de padrões de desenvolvimento de software voltados para esse paradigma e infraestrutura, os quais buscam uma implementação mais purista envolvendo quesitos como redução do uso de implementações multiparadigmas, simplificação no desenvolvimento, garantia de determinismo, entre outras. De modo geral, as novas funcionalidades/recursos para a concepção de aplicações através do *Framework* PON facilitaram e melhoraram seu uso.

Palavras-chave: Paradigma Orientado a Notificações. Padrões de Projeto. Padrões de Implementação. Padrões de Desenvolvimento PON. Roteiro para Concepção de Aplicações PON.

ABSTRACT

RONSZCKA, Adriano F. Contribution for the conceiving of applications in the Notification Oriented Paradigm (NOP) under the vias of patterns. 2012. 236 f. Dissertação (Mestrado em Engenharia de Computação) – Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial,Universidade Tecnológica Federal do Paraná. Curitiba, 2012.

The original materialization of Notification Oriented Paradigm (NOP) previously implemented in the C++ programming language allowed the creation of applications under the domain of this paradigm. Although these contributions to the paradigm, the development of NOP applications still has a low level of maturity and a certain degree of difficulty. Such difficulties arise essentially from the new form of structuring programs, in which such applications follow a notification flow, which differs from the conventional programming. Moreover, so far, only a few applications have been developed, with relatively small scope, which does not contribute effectively to the consolidation of the paradigm, and particularly for learning through an effective set of examples. Henceforth, firstly, the general structure of the framework has been reread and reintroduced under the bias of design patterns, even as there were contributions in its implementation. Subsequently, the main goal of this paper is to guide the development of applications based on NOP in a more streamlined and efficient manner, using the advances mentioned here. This was achieved particularly by the presentation of technical software implementations aimed to this paradigm and its infrastructure, which pursues a manner to conceive more purist implementations, involving issues such as reducing the use of multiparadigms implementations, simplifying the development, ensuring determinism, and so far. In general, the new features for conceiving applications based on the NOP Framework made easier and better its use.

Keywords: Notification-Oriented Paradigm. Design Patterns. Implementation Patterns. NOP Development Patterns. Roadmap for Conceiving NOP Applications.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

CRUD	*Create Retrieve Update Delete*
DIP	*Dependency-Inversion Principle*
DON	Desenvolvimento Orientado a Notificações
FBE	*Fact Base Element*
FIFO	*First-In First-Out*
IDE	*Integrated Development Environment*
ISP	*Interface Segregation Principle*
LIFO	*Last-In First-Out*
LSP	*Liskov Substitution Principle*
MDA	*Model-Driven Architecture*
MVC	*Model View Controler*
OCP	*Open/Closed Principle*
OO	Orientação a Objetos
PC	*Personal Computer*
PD	Paradigma Declarativo
PF	Paradigma Funcional
PI	Paradigma Imperativo
PL	Paradigma Lógico
PP	Paradigma Procidimental
POE	Paradigma Orientado a Eventos
PON	Paradigma Orientado a Notificações
POO	Paradigma Orientado a Objetos
RAM	*Random Access Memory*
RMI	*Remote Method Invocation*
RUP	*Rational Unified Process*
SBR	Sistemas Baseados em Regras
SRP	*Single-Responsibility Principle*
UML	*Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Pesquisas realizadas nas últimas décadas contribuíram significativamente para a evolução tecnológica, especialmente no setor de *software*. Neste sentido, a capacidade de processamento computacional tem crescido em função da evolução das tecnologias [KEYES, 2006]. Entretanto, a utilização da capacidade plena de cada processador nem sempre é devidamente aproveitada em função de limitações das técnicas de programação usualmente empregadas [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; GABBRIELLI e MARTINI, 2010; SIMÃO *et al.*, 2012a].

Na verdade, técnicas de programação baseadas no estado da arte, como o chamado Paradigma Orientado a Objetos (POO), classificado como sendo um subparadigma do Paradigma Imperativo (PI) ou os Sistemas Baseados em Regras (SBR), englobados pelo Paradigma Declarativo (PD), sofrem de limitações intrínsecas de seus paradigmas [BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

Particularmente, esses paradigmas levam ao forte acoplamento de expressões causais e a processamento desnecessário por motivos como redundâncias em avaliações causais e/ou utilização de custosas estruturas de dados para tal. Tais limitações frequentemente comprometem o desempenho pleno das aplicações. Neste âmbito, existem motivações para buscas de alternativas aos PI e PD, com o objetivo de eliminar ou diminuir as desvantagens desses [ROY e HARIDI, 2004; BANASZEWSKI *et al.*, 2007; SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; GABBRIELLI e MARTINI, 2010].

Nesse âmbito, uma alternativa é o chamado Paradigma Orientado a Notificações (PON). A base do PON foi inicialmente proposta por J. M. Simão como uma solução de controle discreto para sistemas de manufatura inteligentes [SIMÃO, 2001; 2005; MANFREDINI *et al.,* 2002; SIMÃO e STADZISZ, 2002; 2008; 2009a; 2009b; SIMÃO, STADZISZ e KÜNZLE, 2003; SIMÃO, STADZISZ e TACLA, 2009; SIMÃO *et al.,* 2002; 2003; 2010]. Posteriormente, essa solução evoluiu para uma solução de controle discreto geral e, então, para uma nova solução de inferência, alcançando por fim a forma de um paradigma de desenvolvimento de *software* [SIMÃO e STADZISZ, 2008; 2009a; SIMÃO *et al.*, 2012a].

Atualmente, o PON se propõe a eliminar algumas das deficiências dos paradigmas usuais de programação em relação a avaliações causais desnecessárias e acopladas, evitando o processo de inferência monolítico baseado em pesquisas ou percorrimentos [SIMÃO e STADZISZ, 2008; 2009a]. Para tal, o PON faz uso de um mecanismo baseado no relacionamento de entidades computacionais que proporcionam um fluxo de execução de maneira reativa através de notificações precisas e pontuais [BANASZEWSKI *et al.*, 2007; SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a]. Além disso, esforços adicionais têm sido produzidos para o estabelecimento desse paradigma [BANASZEWSKI *et al.*, 2007; BANASZEWSKI, 2009; BATISTA *et al.*, 2011; LINHARES *et al.*, 2011; RONSZCKA *et al.*, 2011; VALENÇA *et al.*, 2011; WIECHETECK, 2011; WIECHETECK *et al.*, 2011; SIMÃO *et al.*, 2012a; 2012b; 2012c; 2012d; PETERS, 2012; VALENÇA, 2012].

O fato de o PON ser uma nova solução que resolveria boa parte dos problemas relacionados ao desenvolvimento de aplicações (*e.g.* redundâncias estruturais e temporais) ameniza o problema como um todo (*i.e.* alto acoplamento e redundâncias em avaliações causais), mas certamente não o elimina por completo. Nesse âmbito, padrões de projeto e de codificação mostram-se como uma prática elegante e eficiente na concepção de bons programas devido a sua natureza de prover simplificação e uma melhor estruturação para as aplicações. Desta forma, pesquisas se fazem pertinentes quanto à aplicação de tais práticas na concepção de *software* no PON, sendo esse o escopo desta dissertação.

Neste capítulo introdutório, a Seção 1.1 detalha os fatores motivadores que justificam a preocupação com a relativa baixa qualidade de *software* desenvolvido atualmente, tocando nas questões de descaso com a codificação, comumente sem padronização, e problemas nos paradigmas de desenvolvimento/programação usuais. A Seção 1.2, por sua vez, apresenta a justificativa para o estudo de padrões e suas aplicações na concepção de programas baseados no PON. A Seção 1.3, particularmente, apresenta os objetivos pretendidos com este trabalho. Por fim, a Seção 1.4 apresenta a organização dos capítulos subsequentes que compõem este trabalho.

1.1 MOTIVAÇÃO

A evolução da indústria de computadores/processadores, ao longo das últimas décadas, resultou no desestímulo por parte dos desenvolvedores de *software* usual a construírem programas capazes de executar eficientemente, utilizando o mínimo de recursos computacionais necessários para tal. Isso se deu justamente pela expansão exponencial[1] da capacidade computacional de tais processadores, compensando de certa forma a falta de maiores cuidados para com a eficiência do *software* desenvolvido [RAYMOND, 2003].

Isso até poderia ser aceitável para projetos ou programas de micro e pequeno porte, mas suscitam certa preocupação quando envolvem artefatos maiores e/ou com recursos computacionais limitados [GABBRIELLI e MARTINI, 2010].

Neste sentido, a Subseção 1.1.1 apresenta maiores detalhes sobre os problemas no desenvolvimento de *software*. A Subseção 1.1.2, por sua vez, apresenta os problemas presentes nos próprios paradigmas de programação usuais. Por fim, a Subseção 1.1.3 apresenta o PON como resposta a alguns desses problemas.

1.1.1 Problemas no desenvolvimento de software

O descuido na programação no tocante a performance até pode ser considerado aceitável na implementação de programas usuais de micro, pequeno e mesmo médio porte que executam em equipamentos robustos e atuais. Em tais equipamentos, a ineficiência do *software* diante de uma relativa baixa quantidade de instruções a serem executadas se tornaria impercebível perante suas altas capacidades de processamento [SIMÃO e STASZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

[1] A Lei de Moore (1965) indica que o número de transistores dos *chips* dobra em um período médio de 18 meses, sem reajustar os custos dos mesmos [KEYES, 2006; BANASZEWSKI, 2009].

Entretanto, ao bem da verdade, os descuidos dos programadores na concepção de *software* podem causar sobrecargas de processamento desnecessárias, implicando no uso ineficiente das capacidades de processamento disponíveis [OLIVEIRA e STEWART, 2006; SIMÃO e STASZISZ, 2008; 2009a; SIMÃO *et al.*, 2012a]. Além do mais, em *softwares* complexos, isso pode inclusive esgotar a capacidade do processador, exigindo um ambiente computacional mais rápido ou, até mesmo, um ambiente computacional distribuído (*e.g. dual core*) [HUGHES e HUGHES, 2003; SIMÃO e STASZISZ, 2008; 2009a; SIMÃO *et al.*, 2012a].

Ainda, os maus hábitos de programação podem não ser aceitáveis também em algumas outras classes de *software*, tais como *software* para sistemas embarcados [SIMÃO e STADZISZ, 2008; 2009a; SIMÃO *et al.*, 2012a]. Em tais sistemas, normalmente são empregados processadores com menor poder computacional, geralmente devido a fatores como necessidade de baixo consumo de energia e custos relativamente baixos para aumentar a competitividade no mercado [WOLF, 2007; SIMÃO e STADZISZ, 2008; 2009a; SIMÃO *et al.*, 2012a].

Neste contexto, de acordo com Barr (2011), a quantidade total de processadores produzidos anualmente e utilizados em plataformas embarcadas é aproximadamente 27 vezes superior ao número de processadores produzidos para *PCs*, *Macs* e estações de trabalho *Unix* [BARR, 2011]. Estima-se que a quantidade produzida de computadores em 2011 foi de cerca de 364 milhões [GARTNER, 2011], número relativamente baixo em comparação com os mais de 10 bilhões de processadores embarcados produzidos em 2010 [BARR, 2011].

A popularização dos sistemas embarcados, principalmente em relação ao desenvolvimento de aplicativos para *smartphones* e *tablets*, motivou profissionais da computação, acostumados com a programação usual para computadores, a aderirem a um novo estilo de programação que geralmente envolve mais cuidados com a qualidade e otimização do código desenvolvido. Entretanto, neste contexto, frequentemente, os desenvolvedores tendem a deixar os padrões e boas práticas de programação em segundo plano, visando apenas ganhos pontuais de desempenho [ZECHNER, 2011].

Tal prática pode, de fato, impactar negativamente na estrutura geral das aplicações. Ademais, isso pode inviabilizar as principais vantagens do estado da arte e da técnica em programação de computadores, citando particularmente o projeto e

a programação orientada a objetos [GABBRIELLI e MARTINI, 2010]. No caso de projeto e código bem elaborados, isso facilitaria a reutilização de código, a possibilidade em trabalhar em um nível mais elevado de abstração, entre outras vantagens presentes na programação orientada a objetos [GABBRIELLI e MARTINI, 2010].

Assim sendo, conforme o contexto há problemas particulares no desenvolvimento de *software*. Em suma, em *software* desenvolvido para plataformas comuns existem problemas relacionados ao mal uso de processamento, enquanto em *software* desenvolvido para plataformas embarcadas existem problemas relacionados à estruturação do código. Ao bem da verdade, não raro, tais problemas coabitam em maior ou menor escala dependendo do contexto.

No caso de projetos maiores, particularmente, existem outros problemas aliados aos já citados. Tais projetos geralmente são integrados por uma equipe de desenvolvimento maior, o que dificultaria a padronização de projeto e do código e, consequentemente, afetaria o entendimento e manutenção subsequente do mesmo, ocasionando um aumento considerável nos custos do ciclo de vida do projeto de um *software*. De acordo com uma pesquisa realizada pela empresa *Cast Software*, reparar linhas de código de um projeto mal estruturado, além de aumentar o tempo de entrega do projeto, tende a custar muito caro e comumente apresenta problemas técnicos [THIBODEAU, 2011].

A preocupação com a qualidade de *software* é pertinente, visto que o número de sistemas que exigem maiores cuidados com o processo de desenvolvimento é grande e cresce continuamente. Neste âmbito, a qualidade do *software* não se resume apenas no melhor aproveitamento dos recursos computacionais existentes, mas também em vários outros fatores como escalabilidade, manutenibilidade e extensibilidade, os quais geralmente envolvem a maior parte dos custos de um projeto [PRESSMAN, 2006].

1.1.2 Problemas dos paradigmas de programação usuais

Além do descaso com a eficiência e estruturação do *software* por parte dos desenvolvedores, há outros fatores que influenciam a qualidade de *software* gerado. Um exemplo seria o emprego ou não de métodos de engenharia de *software*. Outro

exemplo, particularmente, seria as deficiências das linguagens de programação utilizadas, tais quais as usuais linguagens imperativas como Pascal/Delphi, C/C++, C#, J#, Java etc. [ROY e HARIDI, 2004; GABBRIELLI e MARTINI, 2010; SIMÃO *et al.*, 2012a].

Em geral, linguagens de programação usuais não apresentam reais facilidades para a concepção de código com módulos otimizados e minimamente acoplados. Isso tende a afetar os custos de processamento, inviabilizar o reuso de módulos e dificultar eventuais distribuições de processamento [GAUDIOT e SOHN, 1990; BANERJEE *et al.*, 1995; RAYMOND, 2003; SIMÃO e STADZISZ, 2009a; SIMÃO *et al.*, 2012a]. Isso se dá devido à estrutura sequencial e a natureza de execução interdependente imposta pelos paradigmas que regem tais linguagens [HUGHES e HUGHES, 2003; BANASZEWSKI *et al.*, 2007; BANASZEWSKI, 2009].

Sucintamente, os paradigmas de programação usuais, mais precisamente o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD), apresentam deficiências que afetam o desempenho das aplicações e a dificuldade na obtenção de "desacoplamento" (ou acoplamento mínimo) entre os módulos de *software* [BANASZEWSKI, 2009; SIMÃO e STADZISZ, 2009a; SIMÃO *et al.*, 2012a].

Essencialmente, o PI impõe buscas orientadas a repetições sobre elementos passivos, relacionando os dados (*e.g.* variáveis) a expressões causais (*i.e.* estruturas *if-then* ou similares). Ademais, as buscas afetam o desempenho das aplicações por gerar redundâncias de processamento e dificultam o alcance de uma dependência mínima entre os módulos pelo fato de gerar acoplamento implícito entre eles [GABBRIELLI e MARTINI, 2010; SIMÃO e STADZISZ, 2009a; SIMÃO *et al.*, 2012a].

O PD, por sua vez, apresenta-se como uma alternativa ao PI. Basicamente, o PD propicia um nível maior de abstração, o que de certa forma, facilita a composição de programas nesse paradigma [KAISLER, 2005; GABBRIELLI e MARTINI, 2010; SIMÃO et al., 2012a]. Além disso, algumas soluções declarativas evitam muitas das redundâncias de execução, a fim de otimizar o desempenho das aplicações, tais como Sistemas Baseados em Regras (SBR) baseados nos algoritmos RETE ou HAL [FORGY, 1982; LEE e CHENG, 2002; SIMÃO *et al.*, 2012a].

No entanto, programas construídos a partir de linguagens usuais do PD (*e.g.* LISP, PROLOG, e SBRs em geral) também apresentam deficiências. Soluções

declarativas fazem uso de estruturas de dados de alto nível computacionalmente custosas, as quais causam consideráveis sobrecargas de processamento [SCOTT, 2000; BANASZEWSKI *et al.*, 2007; SIMÃO e STADZISZ, 2008; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

Assim, mesmo com código redundante, soluções baseadas no PI normalmente apresentam melhor desempenho do que as soluções baseadas no PD [SCOTT, 2000; BANASZESWKI, 2009; SIMÃO *et al.*, 2012a]. Além disso, tal qual na programação baseada no PI, a programação no PD também gera acoplamento entre os módulos, uma vez que o processo de inferência também se baseia em buscas sobre entidades passivas [SIMÃO e STADZISZ, 2008; 2009a; GABBRIELLI e MARTINI, 2010; SIMÃO *et al.*, 2012a]. Ainda, outras abordagens como as baseadas em eventos e programação funcional, mesmo amenizando alguns desses problemas, não os resolvem por completo [SCOTT, 2000; SIMÃO *et al,.* 2012].

Na realidade, de fato, ainda existem questões em aberto sobre o desenvolvimento de *software* no que se refere à facilidade de composição de código otimizado e desacoplado. Novas soluções que simplifiquem a tarefa de construir *software* com tais características são desejáveis. Neste contexto, a solução de programação chamada de Paradigma Orientado a Notificações (PON) foi proposta visando resolver os problemas destacados [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

1.1.3 Paradigma Orientado a Notificações

Em linhas gerais, o chamado Paradigma Orientado a Notificações (PON) resolve certos problemas existentes nos paradigmas usuais de programação, nomeadamente o Paradigma Declarativo (PD) e o Paradigma Imperativo (PI). Na verdade, o PON unifica as principais características e as vantagens do PD (*e.g.* representação do conhecimento em regras) e do PI (*e.g.* flexibilidade de expressão e nível apropriado de abstração), resolvendo, em termos de modelo, várias de suas deficiências e inconveniências em aplicações de *software*, supostamente desde ambientes monoprocessados a completamente multiprocessados [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

O PON permite desacoplar expressões causais do código-fonte, ao considerar cada uma dessas e seus fatos relacionados como entidades computacionais (objetos de *software* nas atuais implementações) e notificantes, o que permite desempenho apropriado e distribuição apropriada (se houver algum multiprocessamento).

Isso é diferente dos programas usuais do PI (salientando os Orientados a Objetos - OO) e do PD (salientado os chamados Sistemas Baseados em Regras – SBR), nos quais expressões causais são passivas e acopladas (senão fortemente acopladas) a outras partes do código, além de haver algum ou mesmo muito desperdício de processamento (conforme o caso) [GABBRIELLI e MARTINI, 2010; SIMÃO *et al.,* 2012a].

No PON, a entidade computacional que trata de uma expressão causal é chamada de *Rule*. As *Rules* gerenciam o conhecimento sobre qualquer comportamento causal no sistema. O conhecimento causal de uma *Rule* provém normalmente de uma regra “se-então”, o que é uma maneira natural de expressão desse tipo de conhecimento. Não obstante, esse conhecimento causal pode ser representado em outro formalismo equivalente quando se fizer pertinente, citando particularmente as chamadas Redes de Petri (RdP) [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; WIECHETECK*,* 2011; WIECHETECK *et al.,* 2011; SIMÃO *et al.*, 2012a].

A Figura 1 apresenta um exemplo de *Rule*, justamente na forma de uma regra causal. Uma *Rule* é composta por uma *Condition* (ou Condição) e por uma *Action* (ou Ação), cf. mostra a figura em questão. A *Condition* trata da decisão da *Rule*, enquanto a *Action* trata da execução das ações associadas. Assim sendo, *Condition* e *Action* trabalham para realizar o conhecimento causal da *Rule*. Na verdade, tanto a *Condition* quanto a *Action* são entidades computacionais agregados à *Rule* [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

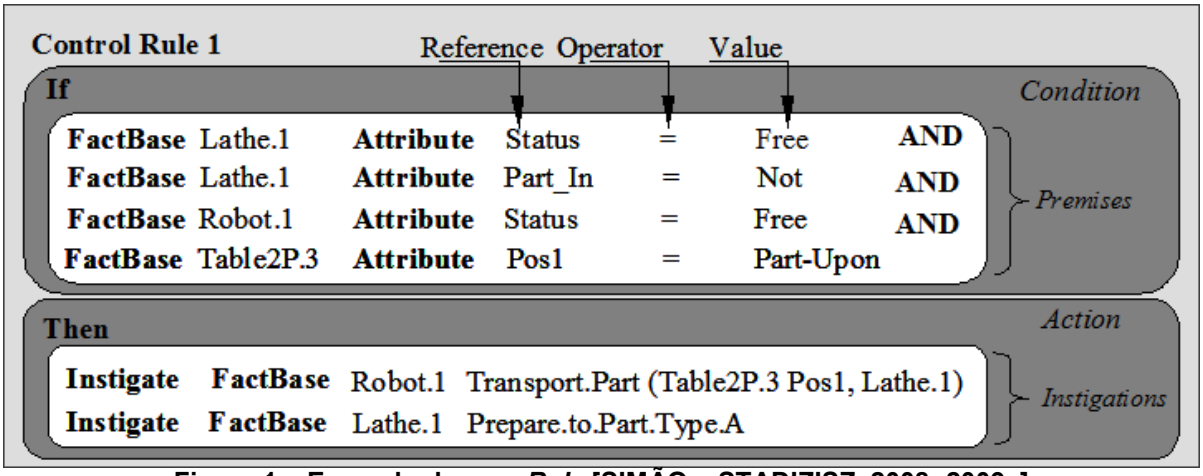


Figura 1 – Exemplo de uma *Rule* [SIMÃO e STADIZISZ, 2008; 2009a]

A *Rule* apresentada na Figura 1 faria parte de um sistema de controle de manufatura avançado, onde equipamentos são tratados a partir de entidades computacionais (*i.e. smart-drivers*). A *Condition* dessa *Rule* lida com a decisão de transporte de peça a partir de uma 'Mesa' (*Smart-Table*) para um 'Torno' (*Smart-Lathe*) utilizando um 'Robô' (*Smart-Robot*). Na verdade, cada uma dessas entidades computacionais, analisáveis por *Conditions*, são chamadas de *Fact Base Elements* (*FBEs*) no PON [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

Conforme ilustrado na Figura 1, a *Condition* daquela *Rule* em questão é composta por três *Premises* (ou Premissas) que se constituem em outro tipo de entidade computacional. Essas *Premises* em questão fazem as seguintes verificações sobre os *FBEs*: a) o Torno está livre e sem peça? b) o Robô está livre? c) há alguma peça na posição 2 da Mesa?. Assim sendo, conclui-se (em geral) que os estados dos atributos dos *FBEs* compõem os fatos a serem avaliados pelas *Premises* [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009].

Na verdade, os estados de cada um dos atributos de um *FBE* são tratados por meio de uma entidade chamada *Attribute* (ou Atributo). Além do mais, e principalmente, para cada mudança de estado de um *Attribute* de um *FBE*, ocorrem automaticamente avaliações (lógicas) somente nas *Premises* relacionadas com eventuais mudanças nos seus estados. Similarmente, a partir da mudança de estado das *Premises*, ocorrem automaticamente avaliações somente nas *Conditions* relacionadas com eventuais mudanças de seus estados [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

Isso tudo se dá por meio de uma elegante cadeia de notificações entre entidades computacionais, cf. ilustra a Figura 2, o que se constitui no ponto central da inovação do PON. Em suma, cada *Attribute* notifica as *Premises* relevantes sobre seus estados somente quando se fizer efetivamente necessário. Cada *Premise*, por sua vez, notifica as *Conditions* relevantes dos seus estados usando o mesmo princípio. Baseado nesses estados notificados é que a *Condition* pode ser aprovada ou não. Se a *Condition* é aprovada, a respectiva *Rule* pode ser ativada executando sua *Action* [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

Uma *Action* também é uma entidade computacional que se conecta a entidades computacionais de outro tipo, as *Instigations* (ou Instigações). No exemplo dado, a *Action* contém duas *Instigations* para: a) ativar o Robô para transportar peças da Mesa (posição 2) para o Torno; e b) preparar o Torno para receber a peça. Efetivamente, o que uma *Instigation* faz é instigar (ativar) um ou mais métodos responsáveis por realizar serviços ou habilidades de um *FBE* [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

Certamente, cada método de um *FBE* é também tratado por uma entidade computacional, que é chamado de *Method* (ou Método). Geralmente, a execução de um *Method* muda o estado de um ou mais *Attributes*. Na verdade, os conceitos de *Attribute* e *Method* representam uma evolução dos conceitos de atributos e métodos de classe do POO. A diferença é o desacoplamento implícito da classe proprietária e a "inteligência" colaborativa pontual para com *Premises* e *Instigations* [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

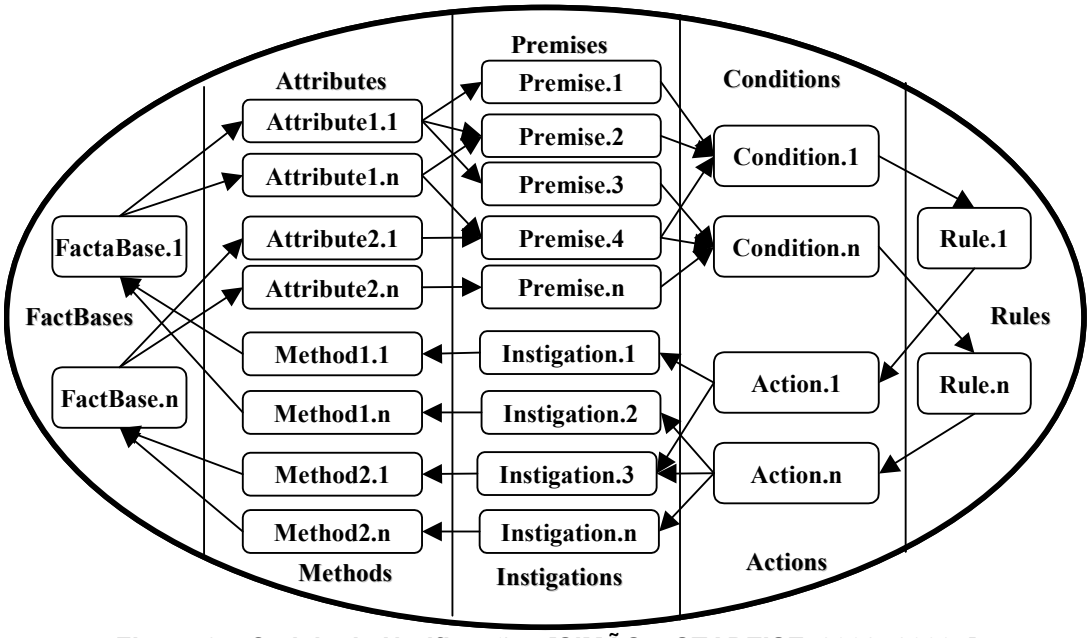


Figura 2 – Cadeia de Notificações [SIMÃO e STADZISZ, 2008; 2009a]

Com isso considerado, salienta-se que a ciência de qual elemento deve notificar outro se dá na própria composição das *Rules*, que pode ser feito em um ambiente amigável na forma de regras causais. Em suma, cada vez que um elemento referenciar outro, o referenciado o considera como elemento a ser notificado quando houver mudanças em seu estado. Por exemplo, quando uma *Premise* faz menção a um dado *Attribute*, esse considera tal *Premise* como elemento a ser notificado. Isso permite emergir o mecanismo ou inferência por notificações sem esforços do desenvolvedor do *software* para tal [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

A natureza do PON leva a uma nova maneira de compor *software*, na qual os fluxos de execução são distribuídos e colaborativos nas entidades. Muito embora o PON permita compor *software* em alto nível na forma de regras, sem o conhecimento da essência do paradigma, tal conhecimento ainda se mostra essencial para a construção de programas mais eficientes, coesos e flexíveis. Por exemplo, é importante conhecer o impacto das notificações entre as entidades PON, uma vez que tais notificações determinam o fluxo de execução de uma aplicação PON. Ademais, aplicações PON apresentam um fluxo de execução não convencional, o que difere o desenvolvimento de aplicações no PON dos demais paradigmas. Assim sendo, o PON permite uma nova maneira de estruturar, executar e pensar os artefatos de *software* [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

No entanto, a concepção de aplicações nesse paradigma leva a dependência de um fluxo de execução bem elaborado, regido principalmente pela composição de um conjunto de *Rules* bem definido. Dependendo da complexidade da aplicação a ser desenvolvida, a definição desse conjunto de *Rules* pode se tornar uma tarefa não-trivial, uma vez que (por exemplo) a responsabilidade de identificar e solucionar conflitos recai sobre esse conjunto de *Rules*. Neste âmbito, estratégias para resolução de conflitos se mostram pertinentes, pois apresentam soluções de resolução de conflitos que automatizam o processo de identificação e o tratamento de conflitos entre o conjunto de *Rules*.

Basicamente, a resolução de conflito consiste na organização das execuções de regras aprovadas segundo alguma estratégia [FRIEDMAN-HILL, 2003; SIMÃO e STADZISZ, 2008; BANASZEWSKI, 2009]. Essas estratégias podem variar para alcançar o fluxo de execução pretendido pelo desenvolvedor tanto em ambientes monoprocessados quanto em ambientes multiprocessados. Ademais, o uso correto de tais estratégias garante, de fato, o determinismo na execução de aplicações desenvolvidas sob os princípios desse paradigma [SIMÃO *et al.*, 2012a].

Outrossim, o PON atualmente está materializado na forma de um *framework/wizard* sob a linguagem de programação C++, enquanto uma linguagem e compilador próprios permanecem como um trabalho futuro. De fato, um paradigma pode ser materializado em outro. Por exemplo, programas orientados a objetos podem ser materializados em linguagem procedimental (inclusive via *framework*) ou programas multiagentes podem ser materializados em linguagem orientada a objetos. Isso é particularmente natural em paradigmas emergentes. Entretanto, seria certamente mais confortável e apropriado um ambiente efetivamente voltado para o PON, o que se constituiria em um trabalho futuro [BANASZEWSKI, 2009].

Neste ínterim, a atual materialização do PON foi comparada em termos de processamento com implementações PI/POO e PD/SBR. No caso de PI/POO houve comparações com programas C++ OO usuais. No caso de PD/SBR, houve comparações com dois *shells*, *CLIPS* e *RuleWorks*, que usam o eficiente motor de inferência *RETE*. Essas comparações apresentaram resultados favoráveis ao PON, ainda que sobre alguns *toy problems* [BANASZEWSKI, 2009].

Também houve outras comparações qualitativas e assintóticas favoráveis ao PON, inclusive em relação a motores de inferência como *RETE*, *TREAT*, *LEAPS* e *HAL* [BANASZEWSKI, 2009]. Contudo, testes em aplicações reais mostraram que,

em alguns casos, a citada materialização do PON seria ligeiramente inferior em termos de desempenho em relação ao POO [BATISTA *et al.*, 2011; LINHARES *et al.*, 2011; RONSZCKA *et al.*, 2011]. No entanto, Valença (2012) com seus esforços de otimização para a melhoria do *framework* PON apresentou resultados favoráveis, com ganhos significativos no desempenho das mesmas aplicações reais apresentadas nos outros artigos. Os ganhos correspondem a um ganho de desempenho de no mínimo 2 vezes. O grande impacto deu-se através das otimizações implementadas sobre o orquestramento das notificações realizadas entre as entidades PON. Isso confirma a afirmação de que a concepção de um compilador próprio para o paradigma é viável e necessário [VALENÇA, 2012].

Apesar da concordância para com Banaszewski (2009) sobre o fato de que as linguagens de programação adotadas e seus respectivos paradigmas influenciam no desempenho das aplicações e que o PON resolve alguns dos problemas de desempenho existentes, o fato é isso não representa por si só a real solução para os problemas suprarrelatados. Salienta-se que os cuidados com o projeto e a programação, em qualquer linguagem e paradigma, devem ser redobrados para que os programas desenvolvidos atinjam o máximo de seu potencial, tanto em questões de desempenho quanto em questões de manutenibilidade e extensibilidade.

Neste sentido, foi criado um método de desenvolvimento para o PON denominado Desenvolvimento Orientado a Notificações (DON) o qual permite elaborar artefatos de projeto voltados particularmente para a Programação Orientada a Notificações segundo um processo que se utiliza do conhecido Projeto Unificado, do ferramental da *Unified Modeling Language* (*UML*) e das pertinentes Redes de Petri [WIECHETECK, 2011; WIECHETECK *et al.*, 2011].

Entretanto, os trabalhos relativos ao DON e ao PON em geral não consideraram efetivamente questões de padrões de desenvolvimento de *software*. Neste sentido, este trabalho busca aliar as qualidades existentes no PON a um estilo de programação eficiente, dotado de boas práticas e padrões de desenvolvimento de *software*. Tal estilo contribuiria para a elaboração de aplicações PON mais eficientes, melhor estruturados, modulares (com unidade coesa e desacopladas) e que possam ser estendidos e reutilizados com certa facilidade.

1.2 JUSTIFICATIVA

Conforme mencionado na Subseção 1.1.3, o PON ameniza o problema de desenvolvimento de *software* (*i.e.* alto acoplamento e redundâncias em avaliações causais), mas certamente não o elimina por completo. Neste âmbito, a maneira com que os módulos de *software* e os relacionamentos existentes entre eles são definidos, impactam diretamente no desempenho e no fluxo de execução de um programa desenvolvido sob os princípios desse paradigma.

Neste sentido, do ponto de vista de Projeto de *Software* voltado ao PON, é possível observar que existe uma preocupação crescente com a organização da estrutura, das partes componentes e dos relacionamentos existentes em um sistema computacional, alcançando o chamado Projeto ou Desenvolvimento Orientado a Notificações (DON) que seria aplicado previamente à Programação Orientada a Notificações [WIECHETECK, 2011, WIECHETECK *et al.*, 2011].

Apesar de tais contribuições para com o paradigma, o desenvolvimento de aplicações no PON ainda apresenta baixo nível de maturidade e certo nível de dificuldade. Tais dificuldades advêm principalmente da nova forma de estruturar os programas, na qual os mesmos seguem um fluxo de notificações, o que difere da programação convencional.

Ademais, até o momento da escrita desta dissertação, apenas algumas aplicações foram desenvolvidas, com escopo relativamente modesto, o que não contribui efetivamente para a consolidação do paradigma e, particularmente, para o aprendizado via um conjunto efetivo de exemplos.

Neste sentido, esta dissertação de mestrado tem como objetivo de estudo nortear o desenvolvimento de aplicações baseadas no PON de maneira mais simplificada e eficiente. Isso se dá pela apresentação de boas práticas de programação em geral, assim como pela apresentação de alguns padrões de projeto e de codificação.

Para isso, primeiramente este trabalho apresenta uma releitura da própria materialização do *Framework* PON, reapresentando-o sob uma nova perspectiva, sob o viés de padrões. Ainda nesse âmbito, contribuições na implementação da estrutura interna desse *framework* foram propostas visando proporcionar melhor

qualidade para esse como um todo, tais como aplicação de padrões de implementação e reestruturação precisa dos padrões de projeto elencados.

Posteriormente, vislumbra-se a possibilidade de explorar novos padrões específicos para o PON, analisando a influência que esses podem causar às aplicações, tanto no impacto em seus desempenhos, quanto em questões de praticidade de desenvolvimento.

Assim sendo, as contribuições deste trabalho devem responder os seguintes questionamentos:

- É apropriado apresentar o PON sob o viés de padrões?
- É possível aplicar e mesmo propor padrões para o PON que se adequem às necessidades dos desenvolvedores e ao mesmo tempo apresentem resultados eficazes? e,
- Como e quando utilizar determinados padrões na concepção de um programa baseado no PON?

As pesquisas apresentadas neste trabalho vão ao encontro dessas questões, apresentando modelos e exemplos desenvolvidos para esse fim.

1.3 OBJETIVOS

Este trabalho possui dois objetivos principais:

(i) Reestruturar e reapresentar o *Framework* PON sob o viés de princípios e padrões de desenvolvimento de *software*; e
(ii) Propor modelos de boas práticas e técnicas de programação na forma de padrões para a concepção de programas no PON.

Para atingir esses objetivos, este trabalho de pesquisa apresenta ainda os seguintes objetivos específicos:

- Estudar os diversos padrões existentes na concepção de *software*;

- Identificar e refinar a aplicação dos padrões existentes na materialização do *Framework* PON original;
- Aplicar outros padrões na materialização (*i.e.* implementação) do *Framework* PON que contribuam para a qualidade das aplicações desenvolvidas sob os princípios do PON;
- Detalhar a estrutura do *Framework* PON sob uma visão orientada a princípios e padrões de projeto;
- Explorar modelos de boas práticas e técnicas de programação na forma de padrões para a concepção de aplicações sob os princípios do PON; e
- Exemplificar e validar a utilização de tais padrões em casos de estudos.

1.4 ORGANIZAÇÃO DO TRABALHO

Este trabalho está descrito em cinco capítulos. No Capítulo 2 são apresentados os conceitos fundamentais que formam a base conceitual utilizada para o desenvolvimento deste. O capítulo apresenta sucintamente os paradigmas usuais de programação e suas derivações, bem como a atual materialização do PON sob o *framework* desenvolvido sob a linguagem de programação C++. Ainda, este capítulo apresenta boas práticas de programação, assim como alguns dos principais padrões de projeto e de implementação existentes.

No Capítulo 3, por sua vez, são apresentados os conceitos do PON de forma detalhada. Mais precisamente, o capítulo explica de forma assaz inovadora a essência desse paradigma sob o viés de padrões. Ainda, neste capítulo são descritos modelos de boas práticas e técnicas de programação na forma de padrões para implementação de programas baseados no PON.

No Capítulo 4, particularmente, são apresentados exemplos práticos da utilização dos modelos propostos com comparações/demonstrações de vantagens e desvantagens de seus usos, tendo com o objetivo nortear os desenvolvedores onde e quando adotar cada uma dessas boas práticas na concepção de programas no PON. Ainda, este capítulo apresenta uma nova implementação de um sistema de pedido de vendas, originalmente considerado em [VENÂNCIA *et al.*, 2011; SIMÃO *et*

al., 2012b], seguindo os modelos propostos, analisando qualitativamente e quantitativamente a solução como um todo.

Por fim, o Capítulo 5 apresenta as conclusões sobre o trabalho desenvolvido, assim como as perspectivas dos possíveis desdobramentos em trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos para estruturar esta dissertação. Primeiramente, a Seção 2.1 apresenta reflexões sobre o atual estado da arte dos paradigmas e linguagens de programação usuais na computação. A Seção 2.2, por sua vez, apresenta o estado da arte do PON em uma revisão mais aprofundada à apresentada na Subseção 1.1.3. Ainda, esta seção detalha a materialização do PON atualmente desenvolvido na linguagem de programação C++. A Seção 2.3 descreve sucintamente o processo de desenvolvimento PON denominado Desenvolvimento Orientado a Notificações. A Seção 2.4, particularmente, apresenta algumas aplicações desenvolvidas, com o intuito de explanar os demais conceitos presentes neste trabalho. A Seção 2.5 contextualiza os principais sintomas de um projeto mal estruturado e os princípios de projeto utilizados com o objetivo de evitar tais sintomas. A Seção 2.6 apresenta os padrões de *software* mais relevantes. A Seção 2.7, particularmente, apresenta as boas práticas de programação e os padrões de implementação mais recomendados na concepção de *software*. Por fim, a Seção 2.8 apresenta as conclusões deste capítulo.

A título de informação, caso o leitor apresente conhecimento sobre as seções deste capítulo, poderá não lê-las ou postergar a leitura sem prejuízo ao entendimento da proposta que é apresentada a partir do Capítulo 3. Não obstante, seria interessante que o leitor dedicasse atenção à Seção 2.4 para um melhor entendimento dos casos de estudo utilizados neste trabalho e, em especial, nas seções 2.6 e 2.7 que apresentam exemplos e conclusões a respeito dos padrões existentes aplicados em tais casos de estudo.

2.1 PARADIGMAS E LINGUAGENS DE PROGRAMAÇÃO

Na ciência da computação, o termo paradigma é empregado como uma maneira de abstrair o pensamento do programador em uma determinada estrutura computacional, capaz de definir o fluxo de execução de um programa. Os paradigmas se diferem em conceitos e abstrações utilizadas para representar os

elementos de um programa (*e.g.* objetos, funções, variáveis, restrições etc.) e a maneira com que esses interagem de maneira a ditar o fluxo de execução de tal programa (*e.g.* atribuições, avaliações causais, repetições, empilhamento, recursividade etc.) [WATT, 2004].

Com o intuito de introduzir tal conceito, esta seção descreve sucintamente a classificação dos paradigmas de programação usuais (Subseção 2.1.1) e apresenta uma reflexão de suas principais deficiências (Subseção 2.1.2).

2.1.1 Classificação dos paradigmas de programação usuais

Os paradigmas de programação usuais (dominantes e emergentes) poderiam ser classificados como subconjuntos de dois paradigmas maiores, o Paradigma Imperativo (PI) e Paradigma Declarativo (PD). O PI pode ser entendido como constituído pelo Paradigma Procedimental (PP) e pelo Paradigma Orientado a Objetos (POO), os quais se diferenciam essencialmente na forma como os elementos e instruções são representados e organizados, sendo o POO considerado mais rico e supostamente estruturado em termos de expressão do código. O PD, por sua vez, pode ser entendido como constituído essencialmente pelo Paradigma Lógico (PL) e pelo Paradigma Funcional (PF). Tais paradigmas se enquadram em um grupo de paradigmas dominantes.

Ainda, os paradigmas de programação emergentes são estabelecidos por meio de um *framework* em uma materialização de um paradigma dominante, o qual forma uma camada intermediária entre os conceitos do paradigma emergente e o paradigma dominante. Ademais, os paradigmas emergentes, em alguns casos, podem ser implementados sob os princípios de diferentes paradigmas dominantes, como no caso do Paradigma Orientado a Agentes. Os paradigmas emergentes também poderiam ser objeto de implementação multiparadigma híbrida [BANASZEWSKI, 2009].

Outrossim, segundo Banaszewski [2009], os subparadigmas que seguem os princípios do PI se caracterizam pela flexibilidade de programação e a forma sequencial pela qual as instruções são executadas pelo mecanismo interno desses paradigmas. Por outro lado, os subparadigmas derivados do PD, diferenciam-se do PI através de um modelo menos flexível, porém mais simplificado de programação,

no qual o programador se concentra na organização do conhecimento sobre a resolução do problema computacional ao invés da implementação propriamente dita.

A facilidade de programação presente no PD representa a principal vantagem desse paradigma em relação ao PI. Porém, para oferecer tal facilidade o PD perde em velocidade de execução para o PI e em certas flexibilidades, principalmente em relação a otimizações algorítmicas e facilidades de acesso ao *hardware* [SCOTT, 2000].

De maneira geral, o modelo de um paradigma está disponível para o desenvolvedor através de uma linguagem de programação. Algumas linguagens de programação foram desenvolvidas para suportar um paradigma específico (*e.g.* Java e Smalltalk suportam o paradigma orientado a objetos, Haskell e Standard ML são baseados no paradigma funcional, enquanto Prolog e Mercury são suportados pelo paradigma lógico), enquanto outras linguagens suportam múltiplos paradigmas (*e.g.* C++, LISP, Leda e Oz) [ROY e HARIDI, 2004].

Uma linguagem de programação multiparadigmas é uma linguagem que prove um arcabouço no qual os programadores livremente podem trabalhar com uma variedade maior de estilos, inter-relacionando estruturas de diferentes paradigmas. O principal objetivo de uma linguagem de programação multiparadigmas é proporcionar aos desenvolvedores uma ferramenta mais eficiente, uma vez que nenhum paradigma por si só fornece a melhor solução para todos os problemas. A utilidade de uma linguagem multiparadigmas depende de quão bem diferentes paradigmas estão integrados [ROY e HARIDI, 2004].

2.1.2 Reflexões sobre deficiências presentes nos paradigmas usuais

Em linhas gerais, tanto o PI quanto o PD apresentam similaridades ao serem baseados em buscas/percorrimentos sobre entidades passivas, as quais consistem em dados (*e.g.* fatos ou estados de variáveis ou de atributos de outras entidades) e comandos de decisão (*e.g.* expressões causais como *se-então* ou regras). Tais buscas afetam o desempenho das aplicações por gerar redundâncias de processamento e acoplamento implícito entre as entidades que compõem uma aplicação [BANASZEWSKI *et al.,* 2007; SIMÃO e STADZISZ, 2008; 2009a; SIMÃO *et al.*, 2012a].

Essencialmente, o PI impõe pesquisas orientadas a laços de repetições sobre elementos passivos, relacionando os dados (*i.e.* variáveis, vetores e árvores) a expressões causais (*i.e.* se-então ou declarações similares). Tais relacionamentos normalmente impactam negativamente nas aplicações, devido sua estrutura monolítica, prolixa e acoplada, o que gera a execução de código não-otimizado e interdependente [BROOKSHEAR, 2006; BANASZEWSKI *et al.,* 2007; SIMÃO e STADZISZ, 2008; GABBRIELLI e MARTINI, 2010; SIMÃO *et al.*, 2012a].

O PD é uma alternativa ao PI. Essencialmente, o PD permite um nível maior de abstração e maior facilidade de programação [KAISLER, 2005; GABBRIELLI e MARTINI, 2010]. Além disso, algumas soluções declarativas podem evitar muitas das redundâncias de execução, a fim de otimizar o processamento. Dentre tais soluções, citam-se os Sistemas Baseados em Regras (SBRs), com base em algoritmos de inferência otimizados como o HAL ou o RETE [FORGY, 1982; CHENG e CHEN, 2000; LEE e CHENG, 2002; KANG e CHENG, 2004]. No entanto, programas construídos com base em linguagens de programação usuais do PD (*e.g. LISP*, *PROLOG* e SBRs em geral) ou mesmo construídos com base em soluções otimizadas (*e.g.* SBRs baseados em RETE) também apresentam desvantagens [BANASZEWSKI *et al.,* 2007; SIMÃO e STADZISZ, 2008; SIMÃO *et al.*, 2012a].

Em verdade, soluções do PD são compostas por estruturas de dados de alto nível, as quais são normalmente caras em termos de processamento. Isso, de fato, agrega em custos de processamento consideráveis, impactando diretamente no desempenho das aplicações. Assim, mesmo com a presença de código redundante, as soluções do PI são normalmente melhores em desempenho do que as soluções do PD [SCOTT, 2000; BANASZEWSKI, 2009]. Além disso, similarmente a programação no PI, a programação no PD também gera código acoplado, devido ao processo similar de inferência baseada em pesquisa sobre entidades passivas [SIMÃO e STADZISZ, 2008; 2009a; GABBRIELLI e MARTINI, 2010]. Ainda, outras abordagens entre o PI e o PD, tais como a programação dirigida por eventos ou a programação funcional, não resolvem tais problemas. Em alguns casos essas reduzem certas redundâncias, em outros apenas atenuam ou fatoram tais problemas [SCOTT, 2000; BROOKSHEAR, 2006; SIMÃO *et al.*, 2012a].

2.1.2.1 Reflexões pontuais da Programação Imperativa

As principais desvantagens da Programação Imperativa estão voltadas para a redundância de código e acoplamento [SIMÃO e STADZISZ, 2009a]. A primeira afeta principalmente o tempo de processamento e a segunda o processo de desacoplamento (e, portanto, o processo de reaproveitamento de módulos/partes e o processo de distribuição de processamento), conforme detalhado a seguir.

A. Redundâncias

Na Programação Imperativa, incluindo a Programação Orientada a Objetos, a presença de código redundante e interdependente é resultado da maneira com que as expressões causais são organizadas e consequentemente avaliadas. Isso é exemplificado no Algoritmo 1 que representa um código habitual e elaborado sem grande esforço técnico e intelectual no POO. Isso significa que o código foi elaborado de uma forma não complicada, como idealmente as aplicações deveriam ser concebidas [SIMÃO e STADZISZ, 2008; BANASZEWSKI, 2009; SIMÃO *et. al*., 2012].

```
1  . . .
2  while (true) do
3    if ((object_1.attribute_1 = 1) and
4        (object_2.attribute_2 = 1) and
5        (object_3.attribute_3 = 1))
6    then
7         object_1.method_1();
8         object_2.method_1();
9         object_3.method_1();
10   end_if
11   . . .
12   if ((object_1.attribute_1 = 1) and
13       (object_2.attribute_n = n) and
14       (object_3.attribute_n = n))
15   then
16        object_1.method_n();
17        object_2.method_n();
18        object_3.method_n();
19   end_if
20 end_while
21 . . .
```

Algoritmo 1 – Exemplo de código redundante na Programação Imperativa [SIMÃO *et al.*, 2012a]

Nesse exemplo, é observado que o laço de repetição força a avaliação (ou inferência) de todas as condições de maneira sequencial. No entanto, a maioria das avaliações é desnecessária, uma vez que somente alguns atributos apresentam alterações em seus estados em cada iteração. Isso até pode ser considerado não importante nesse exemplo simples e pedagógico, sobretudo se o número (n) de expressões causais for pequeno. Entretanto, se for considerado um sistema complexo, integrando muitas partes como aquela, pode-se ter uma grande diferença de desempenho. Ademais, esse tipo de código apresenta os problemas chamados de redundância temporal e estrutural [PAN *et al.*, 1998; SIMÃO e STADZISZ, 2008; 2009a; SIMÃO *et al.*, 2012a].

A redundância estrutural ocorre quando uma expressão lógica não é compartilhada (ou, mais precisamente, quando seu valor booleano não é compartilhado) entre outras expressões causais pertinentes, causando reavaliações desnecessárias. A redundância temporal, por sua vez, ocorre quando uma avaliação lógico-causal é realizada repetidas vezes sobre um elemento já avaliado e inalterado. De fato, ambos os tipos de redundâncias estão presentes no código exemplo apresentado, o qual poderia ser otimizado via esforço de programação adicional. Isto, em escala, torna a programação no PI dificultosa, o que se constitui em outro problema [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009].

B. Acoplamento

Além das usuais avaliações repetitivas e desnecessárias no código imperativo, os elementos avaliados em expressões causais são passivos, embora eles sejam essenciais nesse processo. Por exemplo, uma dada declaração *if-then* ou se-então (ou seja, uma expressão causal) e as variáveis (ou seja, elementos avaliados) não tomam parte na decisão com relação ao momento em que devem ser avaliados [SIMÃO e STADZISZ, 2008; 2009a; SIMÃO *et al.*, 2012a].

A passividade das expressões causais e seus respectivos elementos definem a forma com que esses são avaliados durante a execução de um programa. As avaliações de tais expressões causais e seus respectivos elementos são realizadas sequencialmente pela linha de execução principal (ou pelo menos em linhas de execução presente em *threads*) de um programa, comumente guiada por meio de um laço de repetição. Como essas expressões causais e seus respectivos

elementos não conduzem ativamente sua própria execução (ou seja, eles são passivos), a sua interdependência não é explícita em cada execução do programa [SIMÃO e STADZISZ, 2009a; SIMÃO *et al.*, 2012a].

Sendo assim, as expressões causais e seus respectivos elementos avaliados, dependem dos resultados das avaliações ou estados de outros elementos. Isso significa que eles são acoplados de alguma maneira e que deveriam ser dispostos conjuntamente, pelo menos, no contexto de cada módulo. Esse acoplamento aumenta a complexidade do código, dificultando, por exemplo, (futuro) reaproveitamento de partes do código ou (eventual) distribuição de uma simples parte do código. Isso faz com que cada módulo, ou até mesmo o programa como um todo, seja entendido como uma entidade computacional monolítica [SIMÃO e STADZISZ, 2009a; SIMÃO *et al.*, 2012a].

C. Dificuldade de Distribuição

Mais particularmente, quando a distribuição (*e.g.* distribuição de processo, processador e/ou distribuição por *cluster*) é pretendida, uma análise de código poderia identificar um conjunto de código menos dependente, o que facilitaria sua posterior divisão e distribuição. No entanto, tal atividade é normalmente complexa devido ao acoplamento existente no código e a complexidade resultante da programação imperativa [BANERJEE *et al.,* 1995; WACHTER *et al.*, 2004; SIMÃO *et al.*, 2012a].

Neste sentido, um *software* bem concebido, composto por módulos minimamente acoplados, dotado de avançados conceitos da engenharia de *software* como aspectos [SEVILLA *et al.*, 2008] e projeto axiomático [PIMENTEL e STADZISZ, 2006], poderia ajudar no processo de distribuição [SIMÃO *et al.*, 2012a]. Ainda, *middlewares* como CORBA e RMI poderiam ser úteis em termos de infraestrutura para alguns tipos de distribuição, caso exista um desacoplamento suficiente entre os módulos de *software* [AHMED, 1998; REILLY e REILLY, 2002; SEVILLA *et al.*, 2008; SIMÃO *et al.*, 2012a].

Apesar desses avanços, a distribuição de cada elemento de código ou até mesmo de cada módulo de código ainda é uma atividade complexa, exigindo esforços de pesquisa [WACHTER *et al.*, 2004; GAUDIOT e SOHN, 1990; TILEVICH e SMARAGDAKIS, 2002; JOHNSTON *et al.*, 2004; SEVILLA *et al.*, 2008]. Neste

âmbito, seriam ainda necessários esforços adicionais para alcançar facilidade de distribuição (*e.g.* distribuição automática, rápida e em tempo real), incluindo melhorias no processo de distribuição como um todo (*e.g.* distribuição balanceada e minimamente interdependente) [SIMÃO e STADZISZ, 2009a; SIMÃO *et al.*, 2012a].

A dificuldade de distribuição é um problema, uma vez que existem contextos em que a distribuição é realmente necessária [COULOURIS *et al.*, 2001; HUGHES e HUGHES, 2003; GRUVER, 2007]. Por exemplo, um dado programa otimizado e que, ainda assim, excede a capacidade de um processador disponível, poderia ter seu processamento dividido em um conjunto de processadores [OLIVEIRA e STEWART, 2006]. Tais características podem ser encontradas em diversas aplicações, citando algumas como planta nuclear [DÍAZ *et al.*, 2007], manufatura inteligente [DEEN, 2003; SIMÃO, 2005; TIANFIELD, 2007; SIMÃO, TACLA e STADZISZ, 2009] e controle cooperativo [KUMAR *et al.*, 2005; SIMÃO *et al.*, 2012a].

Além disso, existem outros aplicativos que são inerentemente distribuídos e precisam de uma distribuição flexível, tais como os de computação ubíqua. Exemplos mais precisos são das redes de sensores e algum controle de produção inteligente [LOKE, 2006; TIANFIELD, 2007]. Ainda, em geral, a facilidade e a correta distribuição seriam esperadas, uma vez que existe uma crescente redução de preços no mercado de processadores e, inclusive, devido aos avanços nos modelos de comunicação de rede [TANENBAUM e STEEN, 2002; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

D. Dificuldade de Desenvolvimento

Além das questões de otimização e problemas de distribuição, o desenvolvimento de programas com a Programação pode ser visto como difícil devido sua sintaxe complicada e uma diversidade de conceitos a serem aprendidos, tais como: ponteiros, variáveis de controle e laços aninhados [GIARRATANO e RILEY, 1993].

O processo de desenvolvimento seria propenso a erros, uma vez que quase todo o código é realizado de forma manual, com base em tais conceitos. Neste contexto, o algoritmo imperativo exemplificado (Algoritmo 1) poderia certamente ser otimizado, no entanto sem facilidades significativas; ainda mais quando sua essência estiver dispersa em um sistema de maior porte [SIMÃO *et al.*, 2012a].

Seria necessário investigar soluções melhores do que aquelas fornecidas pelo PI. A solução para resolver alguns de seus problemas pode ser o uso de linguagens de programação de outros paradigmas, como a Programação Declarativa, a qual automatiza o processo de avaliação de expressões causais e seus elementos [ROY e HARIDI, 2004; SIMÃO *et al.*, 2012a]. Ainda, essa abordagem proporciona abstrações que minimizam a realização de algumas tarefas (*i.e.* programa-se "o que fazer" ao invés de "como fazer") [BANASZEWSKI, 2009].

2.1.2.2 Reflexões pontuais da Programação Declarativa

Um exemplo bem conhecido da Programação Declarativa e sua natureza é Sistema Baseado em Regras (SBR) [GIARRATANO e RILEY, 1993; SIMÃO e STADZISZ, 2009a]. Os SBRs provêm uma programação em alto nível baseado na composição de regras causais, o que minimiza o contato dos desenvolvedores com particularidades algorítmicas [GIARRATANO e RILEY, 1993]. Os SBRs são compostos por três entidades modulares genéricas (Base de Fatos, Base de Regras e Motor de Inferência), as quais possuem responsabilidades distintas. Na verdade, essa forma é usual em linguagens declarativas (*i.e.* LISP, PROLOG e CLIPS) [RUSSEL e NORVIG, 2003; SIMÃO *et al.*, 2012a].

Na Programação Declarativa, os estados das variáveis são tratados em uma Base de Fatos e o conhecimento causal em uma Base Causal (ou Base de Regras na programação SBR), as quais são automaticamente combinadas por meio de um Motor de Inferência [GIARRATANO e RILEY, 1993; KANG e CHENG, 2004]. Além disso, alguns algoritmos de inferência (*i.e.* RETE [FORGY, 1982; CHENG e CHEN, 2000; LEE e CHENG, 2002; KANG e CHENG, 2004], TREAT [MIRANKER, 1987; MIRANKER e LOFASO, 1991], LEAPS [MIRANKER *et al.*, 1990] e HAL [LEE e CHENG, 2002]) evitam a maioria das redundâncias temporais e estruturais [BANASZEWSKI, 2009]. No entanto, as estruturas de dados utilizadas para resolver esses problemas implicam em muito consumo de capacidade de processamento [FORGY, 1982; SIMÃO, *et. al*., 2012].

Na verdade, o uso da Programação Declarativa só compensa quando o *software* em desenvolvimento apresenta numerosas redundâncias e pouca variação de dados. Além disso, em geral, um motor de inferência relacionado a uma

determinada linguagem declarativa limita a criatividade do desenvolvedor, o que dificulta algumas otimizações algorítmicas e obscurece o acesso ao *hardware*, o que pode ser inadequado em determinados contextos [SCOTT, 2000; WATT, 2004; BROOKSHEAR, 2006; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012].

A solução para esses problemas pode ser a simbiose entre a programação Declarativa e a Imperativa [ROY e HARIDI, 2004; WATT, 2004]. Na verdade, tal abordagem foi proposta em soluções como *CLIPS*++ [GIARRATANO e RILEY, 1993] e *ILOG Rules* [ALBERT, 1994]. No entanto, tais soluções não são populares devido a fatores como a mistura de sintaxe, mistura de paradigmas e razões técnico-culturais [BANASZEWSKI, 2009]. De qualquer forma, mesmo a Programação Declarativa se apresentando como uma solução relevante, ela não resolve todos os problemas [SIMÃO e STADZISZ, 2008; 2009a; SIMÃO *et al.,* 2012a].

De fato, além da sobrecarga de processamento, a Programação Declarativa também apresenta acoplamento em seu código. Similarmente a Programação Imperativa, programas declarativos possuem um fluxo de execução ou política de inferência, cuja essência é uma entidade monolítica (*i.e.* uma máquina ou motor de inferência). Tal política de inferência é responsável por analisar todos os dados passivos (base de fatos) e inferir a partir do estado desses as expressões causais (regras) afetadas por tais estados. Assim, a inferência baseada em técnica de pesquisa (*i.e. matching*) implica em forte dependência entre a base de fatos e a base de regras [SIMÃO e STADZISZ, 2009a; SIMÃO *et al.*, 2012a].

2.1.2.3 Reflexões pontuais sobre outras abordagens de programação

Melhorias no contexto do PI e do PD têm sido aplicadas com o intuito de reduzir os efeitos de códigos baseados em pesquisas redundantes, tais como a Programação Orientada a Eventos (POE) e a Programação Funcional (PF) [RUSSEL e NORVIG, 2003; FAISON, 2006; BANASZEWSKI, 2009]. A POE e a PF têm sido usadas na concepção de diferentes tipos de *software*, como controle discreto, interfaces gráficas e sistemas multiagentes [RUSSEL e NORVIG, 2003; FAISON, 2006; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

Essencialmente, na POE, cada evento (*e.g.* um botão pressionado, uma interrupção de *hardware* ou uma mensagem recebida) desencadeia uma dada

execução (*i.e.* procedimento, processo ou método), geralmente em um tipo determinado de módulo (*i.e.* bloco, objeto ou até mesmo agente), ao invés de análises sucessivas e repetidas das expressões condicionais para a sua execução. O mesmo princípio se aplica à chamada PF, cuja diferença estaria nas chamadas de funções através de outras funções, em substituição aos eventos. Ainda, função nesse contexto significaria procedimento, método ou alguma unidade similar. Outrossim, programação funcional e orientada a eventos utilizadas em conjunto seria algo usual [SIMÃO *et al.*, 2012a].

No entanto, o algoritmo em cada processo, método ou função é constituído usando a Programação Declarativa ou Imperativa. Isso implica nas deficiências encontradas nesses estilos de programação, como redundância de código, acoplamento etc. De fato, se cada módulo possuir uma considerável quantidade de código causal, eles podem se apresentar como um problema quando em conjunto, tanto em termos de mau uso de processamento quanto em termos de dificuldade de distribuição [SIMÃO *et al.*, 2012a].

Outrossim, uma abordagem alternativa de programação é a chamada Programação Orientada a Fluxo de Dados [JOHNSTON *et al.*, 2004], que supostamente deveria permitir a execução do programa orientada por dados, ao invés de uma linha de execução com base na pesquisa sobre os dados. Portanto, isso facilitaria o desacoplamento e a distribuição [JOHNSTON *et al.*, 2004; SIMÃO *et al.,* 2012a].

Na verdade, a distribuição da Programação Dirigida por Fluxo de Dados é obtida no processamento aritmético, porém não é realmente alcançada em processamento do tipo lógico-causal [GAUDIOT e SOHN, 1990; JOHNSTON *et al.*, 2004]. Esse processamento seria realizado por intermédio de avançados motores de inferência, tais como *RETE* [GAUDIOT e SOHN, 1990; TUTTLE e EICK, 1992; SIMÃO *et al.*, 2012a]. O fato é que os motores de inferência atuais tentam alcançar uma abordagem orientada a fluxo de dados. No entanto, o processo de inferência ainda se baseia em pesquisas, mesmo que se utilizando de assaz otimizadas “árvores” ou grafos de dados. Sendo assim, os problemas relatados persistem [SIMÃO *et al.*, 2012a].

2.1.2.4 Reflexões sobre melhorias na programação

Em suma, conforme explanado, a Programação Imperativa e Declarativa não alcançam de maneira fácil e conjunta alguns requisitos, como a facilidade de obtenção de código otimizado, facilidade e flexibilidade na composição de programas, divisão de código/módulos e distribuição balanceada. Isto é um problema, principalmente quando se considera a demanda de mercado por *software,* onde facilidade de desenvolvimento, código otimizado e processos de distribuição são requisitos atuais [SOMMERVILLE, 2004; PAES e HIRATA, 2008; WATSON *et al.*, 2009]. Na verdade, tal demanda por *software* estimula novas pesquisas e soluções para tornar mais simples a tarefa de construir *software* com tais requisitos [SIMÃO *et al.*, 2012a].

Neste contexto, um novo paradigma de programação chamado Paradigma Orientado Notificação foi proposto para resolver alguns dos problemas apontados [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012a].

2.2 PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON)

Em linhas gerais, o Paradigma Orientado a Notificações (PON) encontra inspirações no PI, tais como flexibilidades algorítmicas e a abstração em forma de classes/objetos do POO e mesmo a reatividade da programação dirigida a eventos. O PON também aproveita conceitos próprios do PD, como facilidade de programação em alto nível e a representação do conhecimento em regras dos SBR. Assim, o PON provê a possibilidade de uso (de parte) de ambos os estilos de programação em seu modelo, ainda que os evolua e mesmo os revolucione (de certa maneira) no tocante ao processo de inferência ou cálculo lógico-causal [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; RONSZCKA *et al.*, 2011; SIMÃO *et al.*, 2012a]. A relação do PON com os paradigmas que inspiraram a sua essência está ilustrada na Figura 3.

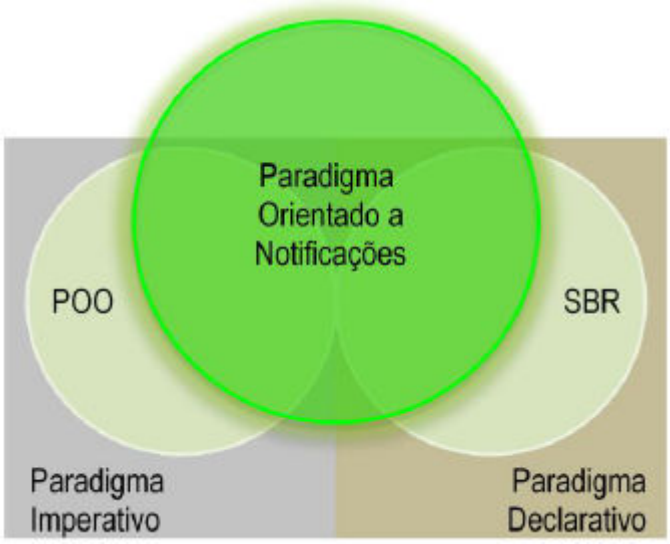


Figura 3 – Relação entre o PON e os paradigmas usuais [BANASZEWSKI, 2011]

Neste âmbito, o PON apresentaria resposta aos problemas desses paradigmas, como repetição de expressões lógicas e reavaliações desnecessárias delas (*i.e.* redundâncias estruturais e temporais) e, particularmente, o acoplamento forte de entidades no tocante às avaliações ou cálculo lógico-causal. Justamente, o PON apresenta outra maneira de realizar tais avaliações ou inferências por meio de entidades computacionais de pequeno porte, ativas e desacopladas que colaboram por meio de notificações pontuais e são criadas a partir do 'conhecimento' de regras [SIMÃO *et al.*, 2012a].

Neste sentido, esta seção detalha o Paradigma Orientado a Notificações (PON), o qual foi introduzido brevemente na Subseção 1.1.3. Mais precisamente, a Subseção 2.2.1 apresenta o mecanismo de notificação do PON. Por sua vez, a Subseção 2.2.2 aborda sobre o mecanismo de resolução de conflitos e garantias de determinismo em aplicações PON. A Subseção 2.2.3, particularmente, reflete e contextualiza sobre as propriedades inerentes ao PON. Ainda, a Subseção 2.2.4 define as características de utilização e compreensão do PON. A Subseção 2.2.5, por sua vez, detalha a função assintótica do PON em relação ao seu processo de resolução do cálculo lógico-causal. Por fim, a Subseção 2.2.6 descreve sucintamente sua materialização por meio de um *framework*.

2.2.1 Mecanismo de Notificações

O Paradigma Orientado Notificações (PON) introduz um novo conceito para a concepção, construção e execução de aplicações de *software*. As aplicações PON são compostas por pequenas entidades reativas e desacopladas, as quais colaboram por meio de notificações precisas e pontuais, ditando assim o fluxo de execução de tais aplicações [SIMÃO e STADZISZ, 2008; 2009a; SIMÃO *et al.*, 2012a]. Essa nova maneira de concepção de *software* tende a proporcionar uma melhora no desempenho das aplicações e, potencialmente, tende a facilitar suas concepções, tanto para ambientes não distribuídos como para ambientes distribuídos [SIMÃO e STADZISZ, 2008; 2009a; SIMÃO *et al.*, 2012a].

O fluxo de iterações das aplicações do PON é realizado de maneira transparente ao desenvolvedor, graças ao orquestramento da cadeia de notificações pontuais entre as entidades PON. Isto é diferente do fluxo de iterações encontrado em aplicações do PI, particularmente do subparadigma OO, onde o desenvolvedor informa de maneira explícita o laço de iteração através de comandos como *while* e *for.* No PON, a repetição ocorre de forma natural na perspectiva de execução da aplicação, conforme exemplificado na Figura 2 e esboçado no diagrama de classes conceitual da Figura 4.

O fluxo de execução ocorre em função da mudança de estado de um objeto *Attribute* de um respectivo *FBE.* Após a mudança de estado do objeto *Attribute*, ele notifica todas as *Premises* pertinentes, para que estas reavaliem seus estados lógicos. Se o valor lógico da *Premise* se altera, a *Premise* colabora com a avaliação lógica de uma ou de um conjunto de *Conditions* conectadas, o que ocorre por meio da notificação sobre a mudança de seu estado lógico a elas. Consequentemente, cada *Condition* notificada avalia o seu valor lógico de acordo com as notificações da *Premise* e com o operador lógico (de conjunção ou disjunção) utilizado. Assim, no caso de uma conjunção, quando todas as *Premises* que integram uma *Condition* são satisfeitas (em estado verdadeiro), a *Condition* também é satisfeita, resultando na aprovação da sua respectiva *Rule* para a execução [BANASZEWSKI, 2009].

Ainda, quando uma dada *Rule* aprovada está pronta para executar (*i.e.* com conflitos resolvidos conforme discute a próxima subseção), a sua *Action* é ativada. Uma *Action* é conectada a um ou vários *Instigations*. Os *Instigations* colaboram com

as atividades das *Actions*, acionando a execução de algum serviço de um objeto *FBE* por meio dos seus objetos *Methods*. Geralmente, as chamadas para os *Methods* mudam os estados dos *Attributes* e o ciclo de notificação recomeça [BANASZEWSKI, 2009].

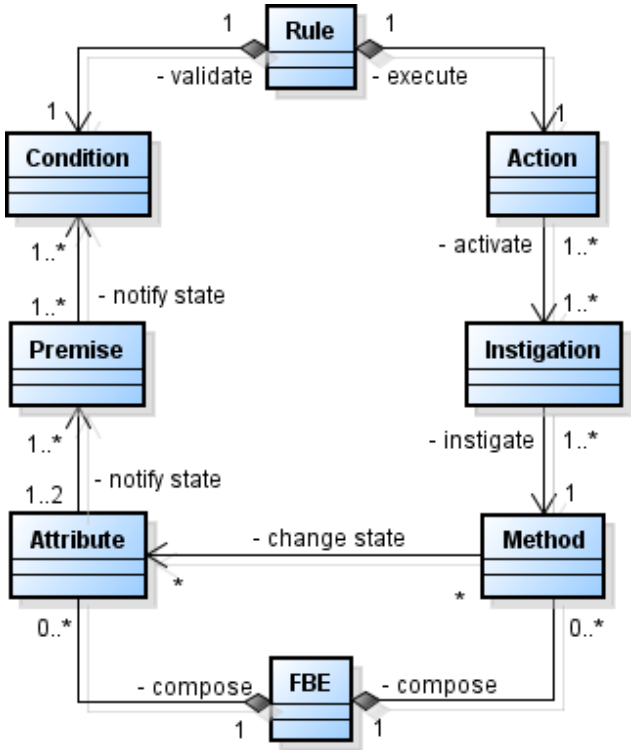


Figura 4 – Principais entidades do PON e seus relacionamentos [BANASZEWSKI, 2009]

Oportunamente, as conexões entre os objetos notificantes são estabelecidas em tempo de criação e por emergência. Por exemplo, na criação de um objeto *Premise* pelo menos um objeto *Attribute* é considerado (como o seu *Reference*). Uma vez que um *Attribute* é referenciado em uma *Premise*, o *Attribute* considera automaticamente esta *Premise* como sendo interessada em receber notificações sobre o seu estado. Assim, o *Attribute* identifica todas as *Premises* interessadas e notifica-as quando o seu estado muda. Ainda, mecanismo similar ocorre em relação às *Premises* e as Conditions, bem como as relações entre *Conditions* e *Rules* [BANASZEWSKI, 2009].

2.2.2 Resolução de Conflitos no PON

Um conflito ocorre quando duas ou mais *Rules* referenciam um mesmo *FBE* e demandam exclusividade de acesso a este *FBE*. Deste modo, as *Rules* concorrem para adquirir acesso exclusivo a este *FBE*, sendo que somente uma destas *Rules* em conflito pode executar por vez, a qual obteve o acesso exclusivo. Neste âmbito, para resolver as questões de resolução de conflitos entre as *Rules*, basicamente o fluxo de sua execução é determinado segundo uma estratégia pré-estabelecida. Essas estratégias podem variar para alcançar o fluxo de execução pretendido pelo desenvolvedor tanto em ambientes monoprocessados quanto em ambientes multiprocessados.

Em um ambiente monoprocessado, a resolução de conflitos ocorre para estabelecer a ordem de execução das *Rules*, onde apenas uma *Rule* pode executar por vez. Em um ambiente multiprocessado, a resolução de conflitos ocorre para evitar o acesso concorrente a um recurso referenciado por várias *Rules* a fim de manter a consistência da aplicação PON [BANASZEWSKI, 2009].

Em se tratando de ambientes monoprocessados basicamente é empregado um escalonador de *Rules* formado por uma estrutura de dados do tipo linear (*e.g.* pilha, fila ou lista) [BANASZEWSKI, 2009]. Essas estruturas guardam referências para as *Rules* aprovadas, conforme ilustra a Figura 5. Assim, tais estruturas recebem as *Rules* na ordem em que elas são aprovadas, podendo reorganizá-las de acordo com os preceitos de cada estratégia adotada [BANASZEWSKI, 2009].

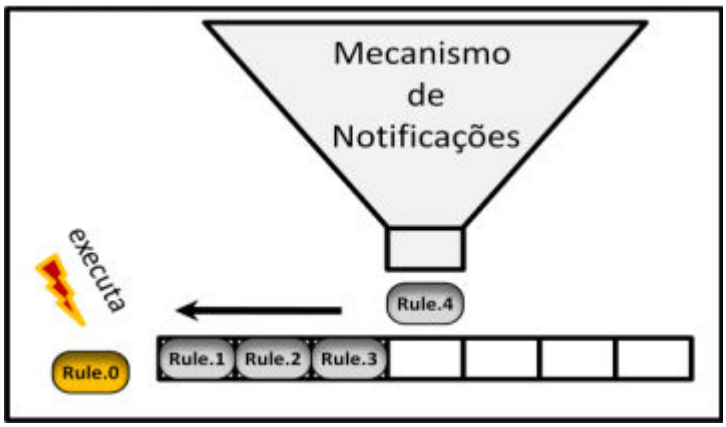


Figura 5 – Modelo Centralizado de Resolução de Conflitos [BANASZEWSKI, 2009]

Desta forma, conforme a estratégia de resolução de conflitos pré-determinada pelo desenvolvedor, as *Rules* em questão serão efetivamente executadas. Neste âmbito, os modelos de resolução de conflitos empregados para o PON em ambientes monoprocessados são:

- ***BREADTH***: se baseia no escalonamento *First In, First Out (FIFO)*, ou seja, refere-se à execução de entidades *Rule*, seguindo uma estrutura de dados do tipo fila*;*
- ***DEPTH***: se baseia no escalonamento *Last in, First Out (LIFO)*, ou seja, refere-se à execução de entidades *Rule*, seguindo uma estrutura de dados do tipo pilha; e
- ***PRIORITY***: organiza as entidades *Rule* de acordo com as prioridades definidas nas mesmas.

Quando nenhuma estratégia for definida pelo desenvolvedor, utiliza-se a estratégia NO_ONE, a qual faz com que as entidades Rules não sejam enviadas ao escalonador/estrutura de dados e sejam aprovadas e executadas imediatamente. Basicamente, a definição do tipo da resolução de conflito adotada pelo desenvolvedor implica no método utilizado para a execução das Rules da aplicação. A definição de resolução de conflitos de Rules deve ser adotada principalmente em aplicações distribuídas e/ou concorrentes.

As soluções para evitar os conflitos apresentados são particularmente aplicáveis a soluções PON monoprocessadas, ainda que até possam ser úteis em soluções multiprocessadas e distribuídas [BANASZEWSKI, 2009]. Entretanto, na verdade, a definição de resolução de conflitos de Rules deve ser adotada em aplicações concorrentes e/ou distribuídas com soluções que lhe sejam mais apropriadas. Neste sentido, ainda que este trabalho não seja relativo à aplicação de PON em sistemas concorrentes ou distribuídos, nestes trabalhos [SIMÃO, 2005; BANASZEWSKI, 2009; SIMÃO e STADZISZ, 2008; 2009a; SIMÃO et. al., 2010] encontram-se soluções úteis ao PON para resolução de conflitos em aplicações PON distribuídas, bem como soluções correlatas para a garantia de determinismo.

Neste sentido, com os conflitos solucionados (e determinismo garantido), uma dada Rule aprovada está pronta para executar o conteúdo da sua Action. Uma Action é conectada a um ou vários Instigations. Os Instigations colaboram com as

atividades das Actions, acionando a execução de algum serviço de um objeto FBE por meio dos seus objetos Methods. Geralmente, as chamadas para os Methods mudam os estados dos Attributes e o ciclo de notificação recomeça [BANASZEWSKI, 2009].

2.2.3 Propriedades inerentes ao PON

Nota-se que a essência da computação no PON está organizada e distribuída entre entidades autônomas e reativas que colaboram por meio de notificações pontuais. Esse arranjo forma o mecanismo de notificações, o qual determina o fluxo de execução das aplicações. Por meio desse mecanismo, as responsabilidades de um programa são divididas entre os objetos do modelo, o que permite execução otimizada e 'desacoplada' (*i.e.* minimamente acoplada) útil para o aproveitamento correto de monoprocessamento, bem como para o processamento distribuído.

Neste sentido, toda essa colaboração por meio de notificações pontuais e precisas representaria a solução para as principais deficiências dos atuais paradigmas de programação. Ao evitar buscas sobre entidades passivas, o PON implicitamente evita as redundâncias estruturais e temporais que tanto afetam o desempenho das aplicações no PI e mesmo no PD [BANASZEWSKI, 2009; SIMÃO *et al.*, 2012a].

Ademais, observa-se que os objetos participantes da cadeia de notificação do PON se apresentam desacoplados, devido à comunicação realizada por meio de notificações pontuais. Neste âmbito, pode-se dizer que aplicações no PON possuem características apropriadas para a execução em ambientes multiprocessados, uma vez que 'somente' se faz necessário os objetos notificantes conhecerem os endereços dos objetos a serem notificados para a inferência por notificação ocorrer [BANASZEWSKI, 2009].

2.2.4 PON - Utilização x Compreensão

A natureza do PON leva a uma nova maneira de desenvolver *software*, onde os fluxos de execução são distribuídos e colaborativos nas entidades. Assim sendo, o PON permite uma nova maneira de estruturar, executar e pensar os artefatos de *software*. Ainda, muito embora o PON permita compor *software* em alto nível na forma de regras sem o conhecimento desta sua essência, conhecê-la é deveras importante [SIMÃO *et al.*, 2012a].

Por exemplo, é importante saber dos impactos de desempenho quanto à aplicação das estratégias de resolução de conflitos e das estratégias de distribuição. Neste último caso, um exemplo preciso seria a forma de agrupar elementos de maior fluxo de notificações em um mesmo 'grupo', evitando assim comunicações desnecessárias nos canais de comunicação (*e.g.* redes) [SIMÃO *et al.*, 2012a].

Ainda, a compreensão dos princípios do PON é importante para aplicações complexas, onde o fluxo de notificações é intenso e precisa de mais formalismo e rastreabilidade, como em aplicações de tempo real e controle discreto. Na verdade, esse tipo de aplicação pode exigir apoio de ferramentas formais para elaboração do projeto [SIMÃO *et al.*, 2012a].

Um exemplo particular de formalismo é a rede de Petri. Na verdade, redes de Petri são compatíveis com os sistemas baseados em regras, em geral, em termos de expressão de relações causais [SHEN e JUANG, 2008]. Além disso, são particularmente compatíveis com os princípios do PON também em termos da sua essência [SIMÃO e STADZISZ, 2009a]. Neste contexto, seria necessário conhecer o PON e os princípios do domínio de rede de Petri, compreendendo que ambos são naturalmente compatíveis [SIMÃO e STADZISZ, 2009a].

2.2.5 Cálculo Assintótico da Inferência do PON

A complexidade assintótica polinomial do PON, no pior cenário, é representada por $O(n^3)$ ou $O(FactBaseSize * nPremises * nRules)$, onde *FactBaseSize* corresponde ao tamanho máximo de objetos *Attributes*, *nPremises* corresponde ao tamanho máximo de objetos *Premises* notificados por estes

Attributes e *nRules* corresponde ao tamanho máximo de objetos *Conditions* notificados por estas *Premises* [SIMÃO, 2005; BANAZEWSKI, 2009].

A função assintótica apresentada para o PON, no pior cenário, demonstra uma solução bastante similar ao mecanismo de notificações do algoritmo *HAL* [BANAZEWSKI, 2009]. Ainda a função temporal polinomial do *HAL*[2] (*O(n³)*) se apresenta mais eficiente do que os algoritmos de inferência *RETE*, *TREAT* e *LEAPS* [BANAZEWSKI, 2009].

Essa função assintótica representa a quantidade de notificações entre os objetos colaboradores que também corresponde à quantidade de avaliações lógicas. A constatação desta função assintótica pode ser realizada pela análise da Figura 6, a qual demonstra as relações por notificações entre os objetos colaboradores. Nesta os *Attributes*, *Premises*, *Conditions* e *Rules* correspondem respectivamente aos símbolos com abreviações *Att, Pr, Cde* e *Rl* [BANAZEWSKI, 2009].

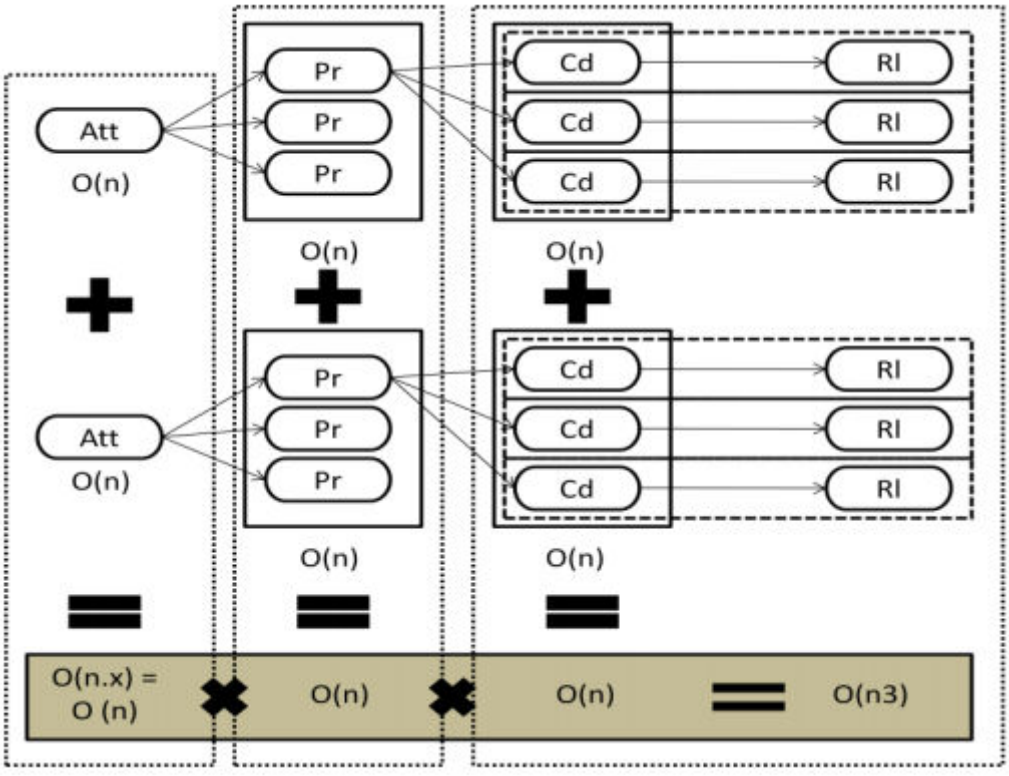


Figura 6 – Cálculo assintótico do mecanismo de notificações [BANAZEWSKI, 2009]

[2] Em relação ao HAL, apesar das grandes semelhanças, o mecanismo de notificações se diferencia na comunicação mais pontual entre os objetos. Enquanto que no HAL somente os componentes genéricos se comunicam, no mecanismo de notificações as próprias instâncias podem ser comunicar e de forma direta, evitando que entidades genéricas despendam buscas para relacionar as instâncias comunicantes [BANAZEWSKI, 2009].

Ademais, outra forma adequada de analisar a complexidade polinomial do PON é considerar o caso médio. A análise da complexidade do caso médio é iniciada analisando-se o começo do processo de notificação do PON através da entidade *Attribute*. Assim, as principais variáveis envolvidas em uma notificação de um *Attribute* é demostrada pela equação da Figura 7.

$$FB_{at}() = NumPremises + NumRules$$

Figura 7 – Complexidade da Notificação *Attribute* [SIMÃO, 2005].

A variável "*NumPremises"* é a soma de entidades *Premises* ao respectivo *Attribute* e a variável "*NumRules"* é a soma das entidades *Rules* a cada entidade *Premise* contada em *"NumPremises".* Portanto, se for considerado simplesmente cada ciclo de inferência como a instigação de um *Attribute*, uma média possível seria: TMedium(x) = (FBAT.1()+ … + FBAT.w()) / w, onde (w) é o número de todos os *Attributes* existentes. Assim, o resultado desta média seria uma ordem de (n), o que implicaria em uma complexidade linear *O(n)* [SIMÃO, 2005].

2.2.6 Materialização do PON

Os conceitos do PON propriamente dito foram primeiramente materializados sobre o POO, através de um arquétipo ou *framework* desenvolvido com a linguagem de programação C++. A versão prototipal concebida por Simão e versão original do *Framework* PON desenvolvida por Banaszewski [2009] foram implementadas especificamente para ambientes monoprocessados, contemplando os conceitos do paradigma PON apresentado na seção anterior. Dado que o original se demonstra melhor que o prototipal, a versão considerada neste trabalho será a original.

Estruturalmente, o *Framework* PON (original) materializa as entidades colaboradoras do paradigma em forma de classes/objetos relacionados através de estruturas de dados com referências às entidades interessadas em seus estados. Neste âmbito, uma estrutura de pacotes foi projetada, conforme Figura 8, para modelar (e explicar) esta materialização do PON.

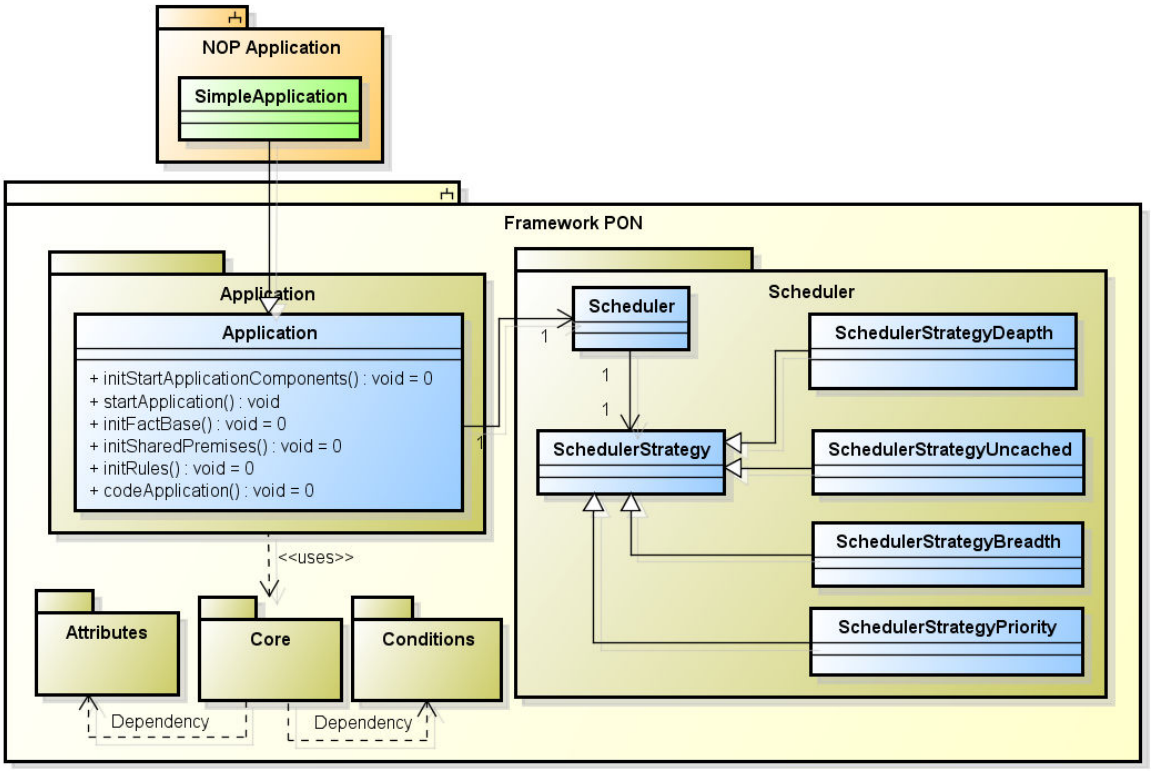


Figura 8 – Estrutura do *framework* do PON

Conforme ilustrado na Figura 8, o *Framework* PON é subdividido em três pacotes principais. Dentre esses, o pacote *Application* é formado exclusivamente pela classe *Application*, que representa a ponte de ligação entre uma aplicação PON e as demais classes do *framework*. O pacote *Scheduler*, por sua vez, representa as implementações das classes de resolução de conflitos, conforme descrito na Subseção 2.2.3. O pacote *Core*, particularmente, é formado pelas classes colaboradoras que realizam o processo de notificação do *Framework* PON. A Figura 9 ilustra o diagrama de classes que melhor ilustra os componentes do pacote *Core*.

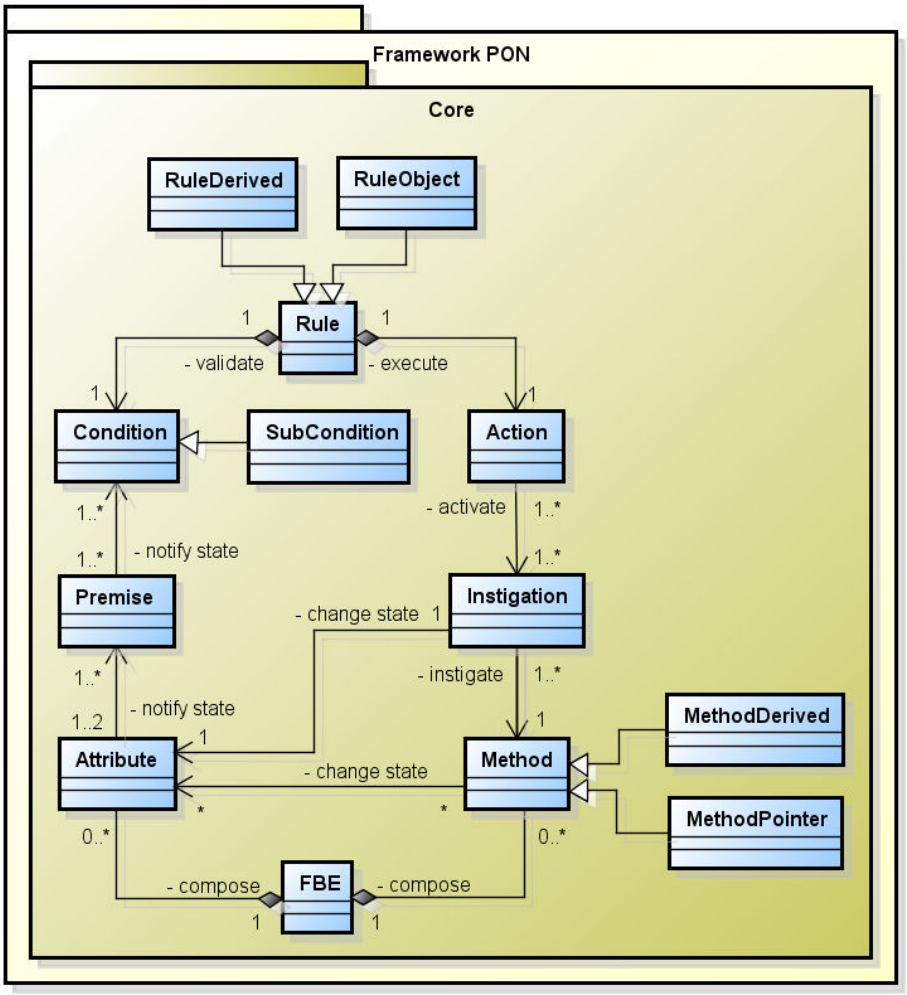


Figura 9 – Estrutura do pacote *Core*

As classes *Rule* e *FBE* se apresentam nas extremidades opostas e se relacionam por meio de suas classes colaboradoras – *Attribute*, *Premise*, *Condition*, *Action*, *Instigation* e *Method* – sendo que a colaboração entre os objetos destas classes determina o fluxo de execução de uma aplicação do PON. Ademais, as classes *Method* e *Rule* que definem as entidades puras do PON são estendidas de modo a proporcionar funcionalidades adicionais.

Ainda, o pacote *Core* é composto também pelos subpacotes *Attributes* e *Conditions*. Conforme apresenta a Figura 10, o subpacote *Attribute* é formado pelas classes responsáveis por encapsular os tipos primitivos do POO. Estas classes (*Boolean, Char, Double, Integer e String*) introduzem reatividade aos tipos primitivos, permitindo que estes façam parte de estruturas causais do PON.

Ademais, conforme a Figura 10, o subpacote *Conditions* é formado pelas classes que fazem parte da composição de uma respectiva *Condition*. Particularmente, a classe *LogicalOperator* e suas derivadas (*Conjunction,*

Disjunction e *Single*), definem a operação lógica utilizada pela *Condition* de maneira a aprovar uma determinada *Rule*.

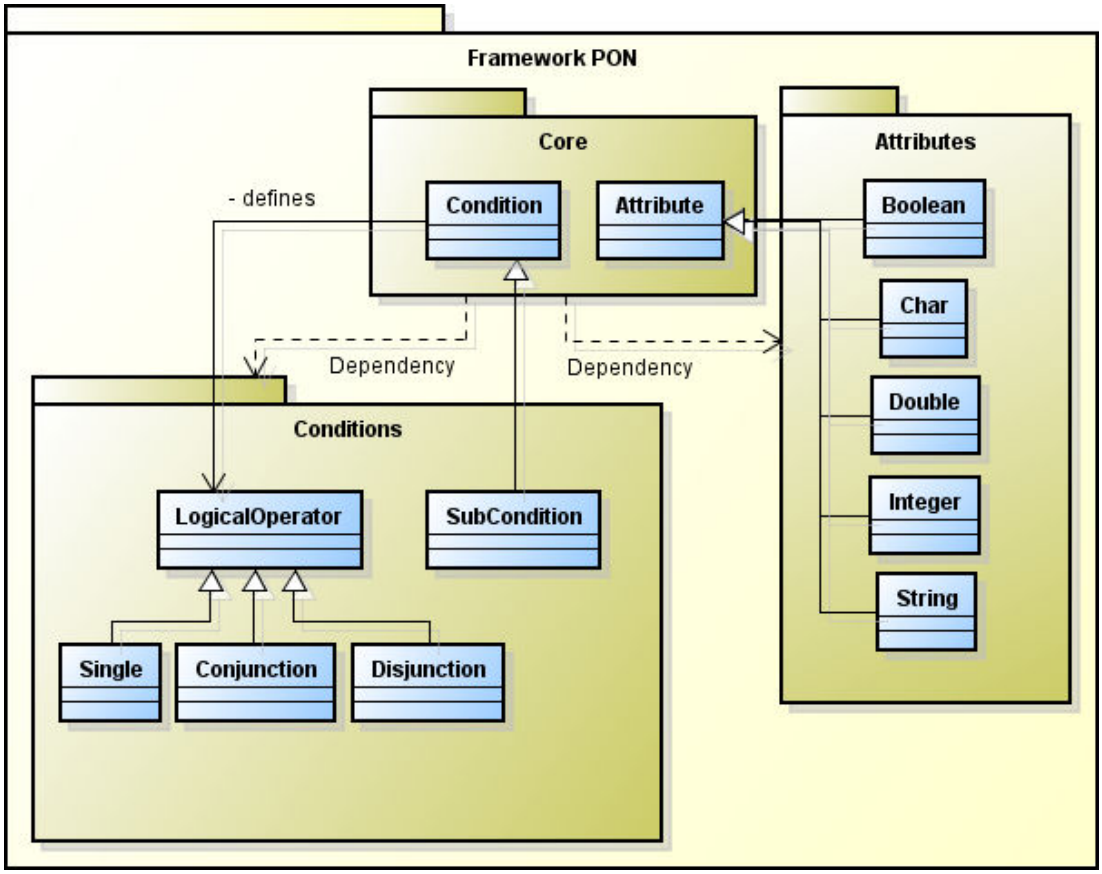


Figura 10 – Estrutura dos subpacotes *Attributes* e *Conditions*

Essa materialização do PON convenientemente possibilitou a validação dos conceitos relacionados a este paradigma, principalmente em questões da resolução do cálculo lógico-causal e assim a eliminação das redundâncias temporais e estruturais. Em testes comparativos apresentados em [BANAZEWSKI, 2009; SIMÃO *et al.,* 2012a], o PON se mostrou já efetivo nesta materialização, entretanto conforme discutido em [BATISTA *et al.,* 2011; LINHARES *et al.,* 2011; RONSZCKA *et al.,* 2011; VALENÇA *et al.,* 2011; SIMÃO *et al.*, 2012b], tal materialização ainda demonstra deficiências em alguns domínios de aplicações.

Os problemas descritos nos trabalhos são referentes principalmente sobre a camada extra de execução das aplicações PON, conforme esboçado na Figura 11.

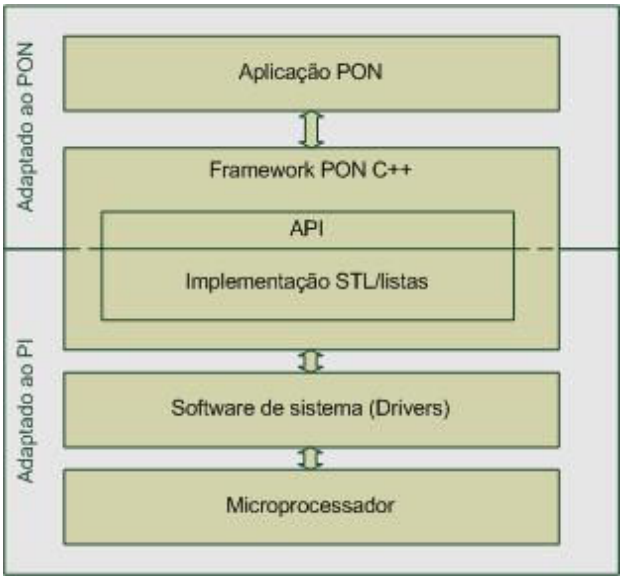


Figura 11 – Camadas da arquitetura do *Framework* PON

É possível observar na Figura 11 que as aplicações concebidas com o auxílio do *Framework* PON são dependentes de camadas extras, o que contribui para a degradação de seus desempenhos. A implementação do *framework*, uma vez desenvolvido sob os princípios da linguagem de programação C++, ainda poderia ter seu âmbito otimizado, algo atualmente em constante desenvolvimento [VALENÇA *et al.,* 2011; VALENÇA, 2012].

Ademais, foi possível observar que não somente otimizações no tocante de desempenho realizadas no *framework* são desejáveis e pertinentes, como também boas práticas e cuidados com a maneira com que as aplicações são concebidas impactariam igualmente no desempenho dessas. Neste sentido, as seções subsequentes visam encontrar técnicas, procedimentos e padrões no sentido de agregar melhorias no âmbito da composição de aplicações sob o viés deste paradigma.

2.3 DESENVOLVIMENTO ORIENTADO A NOTIFICAÇÕES

O método denominado Desenvolvimento Orientado a Notificações (DON) foi concebido pelo trabalho realizado por WIECHETECK (2011). O método é empregado em projetos de *software* baseados em desenvolvimento de aplicações PON implementados a partir do *Framework* PON. O Método DON é compatível com

as práticas atuais de engenharia de *software*, porém apresenta técnicas específicas de modelagem orientadas ao desenvolvimento de *softwares* no PON.

Neste âmbito, a criação do método de desenvolvimento em questão requereu duas etapas: (1) a criação de um perfil em *Unified Modeling Language* (*UML*) denominado *Perfil* PON, que define os principais conceitos do PON por meio da utilização de mecanismos de extensão da *UML*; e (2) a criação do método DON, propriamente dito, que faz uso do *Perfil* PON (*NOP Profile*) e apresenta uma sequência de passos para a construção de projetos no PON. As próximas subseções detalham os dois passos supracitados.

2.3.1 Perfil *UML* para o PON - *NOP Profile*

De maneira sucinta, o perfil *UML* para o PON propõe a extensão do metamodelo da *UML,* a qual permite uma nova sintaxe e semântica aos seus elementos constituintes. Um conjunto desses elementos, agrupados dentro de um pacote, formam um Perfil *UML*. Os mecanismos que permitem extensão de novos elementos de modelagem ao metamodelo da *UML* são [LIMA, 2008]:

- **Estereótipo**: permite a classificação de um elemento de modelo de acordo com um elemento de modelo base já existente na *UML*. Estereótipos devem possuir restrições e valores etiquetados para adicionar informações necessárias aos novos elementos (ex: *idClass* na Figura 12);
- **Valor etiquetado**: permite explicitar uma propriedade de um elemento, sendo que essas informações podem ser adicionadas a qualquer elemento do modelo. No metamodelo, os valores etiquetados são representados como atributos da classe que define um estereótipo (ex: identificador na Figura 12), mas no modelo são apresentadas em um novo compartimento dos elementos, denominado “*tags*”.
- **Restrição**: é uma informação semântica anexada a um ou mais elementos do modelo para expressar uma condição que o sistema deve satisfazer. Tal especificação é escrita em uma determinada linguagem de restrições, conforme observado na Figura 12.

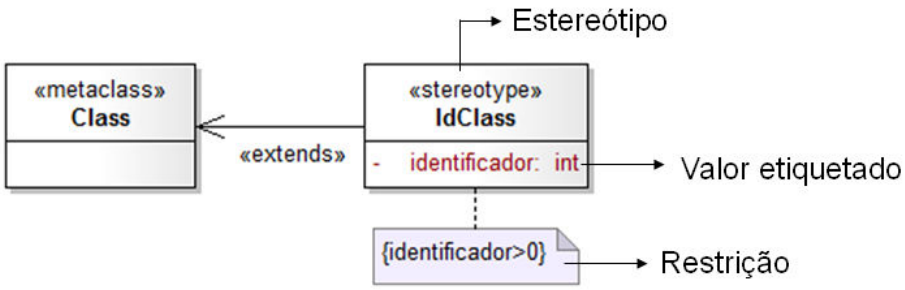


Figura 12 – Mecanismo de extensão da *UML* [WIECHETECK *et al.*, 2011]

Neste âmbito, o mecanismo de extensão da *UML* foi atribuído às entidades participantes do modelo de domínio de uma aplicação PON. Um modelo de domínio pode ser construído usando-se dos elementos de modelagem da *UML* como classes, pacotes e associações [WIECHETECK *et al.*, 2011]. Assim, para a concepção de aplicações PON, as entidades participantes do modelo de domínio pertencem basicamente às classes que fazem parte dos pacotes *Application* e *Scheduler* esboçado pela Figura 8 e do pacote *Core* esboçado pela Figura 9.

Após a identificação dos modelos do domínio do PON, o próximo passo foi a criação do perfil *UML* para o PON, denominado o *NOP Profile*. O *NOP Profile* é composto por um pacote *UML* estereotipado (<<*profile*>>) com mesmo nome do perfil, que compreende outros dois pacotes – *NOP Profile Core* e *NOP Profile Application* – um para cada pacote de classes do *Framework* do PON [WIECHETECK *et al.*, 2011].

O *NOP Profile Core* é composto por elementos de extensão da *UML* (estereótipos, valores etiquetados e restrições) que representam e descrevem os objetos participantes do mecanismo de notificações do PON. Esses elementos de extensão foram obtidos a partir da análise do modelo do domínio da Figura 9 (que ilustra o diagrama de classes do *Framework* PON do pacote *core*). Primeiramente, foi criado um estereótipo para cada elemento relevante definido no modelo do domínio e, em seguida, foram associados esses estereótipos aos elementos do metamodelo da *UML* por meio do relacionamento de extensão (<<*extends*>>). A Figura 13 exibe o perfil *NOP Profile Core* criado [WIECHETECK *et al.*, 2011].

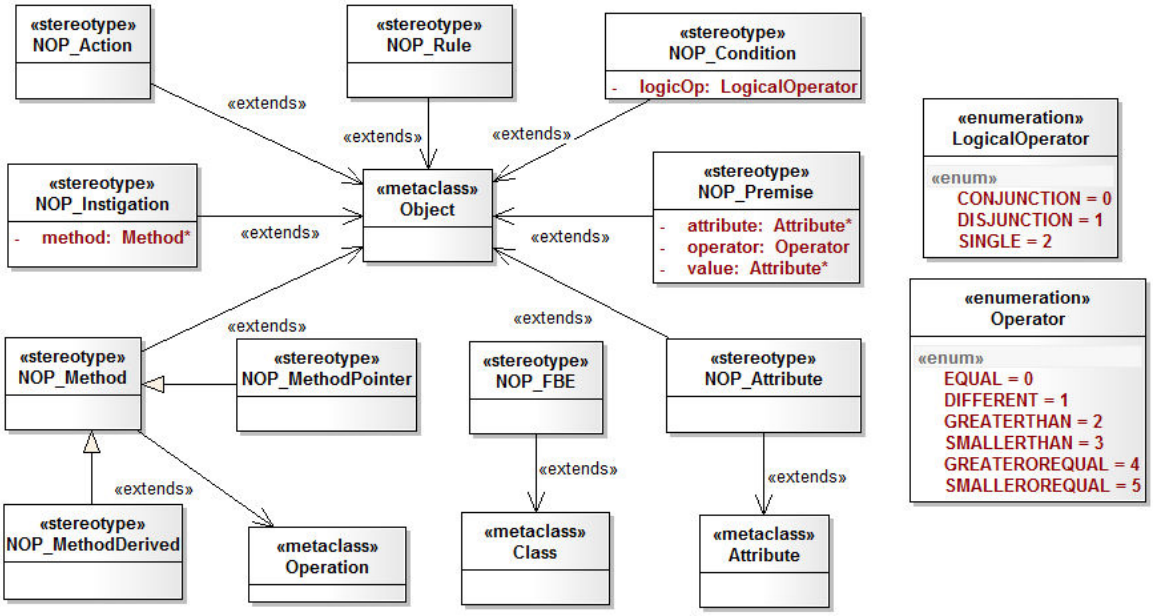


Figura 13 – NOP Profile pacote core [WIECHETECK *et al.*, 2011]

Seguindo os mesmos passos para a criação do *NOP Profile Core*, foi construído o *NOP Profile Application* analisando-se o modelo de domínio da Figura 8. Neste perfil, a classe *Application* do modelo do domínio foi transformada em um estereótipo – *NOP_Application* – que estende a metaclasse *Class* do metamodelo da *UML*, uma vez que uma nova aplicação no PON é representada como uma subclasse da classe *Application*. Já os métodos da classe *Application* do *Framework* PON foram transformados em estereótipos que estendem a metaclasse *Operation* da *UML*, sendo eles: *initFactBase* e *initRules*. Também foi criada uma enumeração – *SchedulerStrategy* – que define as estratégias de resolução de conflitos entre regras existentes no PON, que podem ser: *BREADTH*, *PRIORITY*, *DEPTH*, *UNCACHED* e *NO_ONE*. A Figura 14 ilustra o pacote de perfil *NOP Profile Application* criado [WIECHETECK *et al.*, 2011].

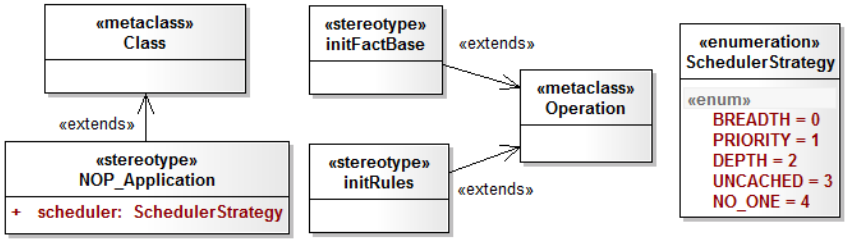


Figura 14 – *NOP Profile Application* pacote *application* [WIECHETECK *et al.*, 2011]

2.3.2 Processo de Desenvolvimento Orientado a Notificações

Após a elaboração do perfil *UML* para o PON (*NOP Profile*), o arquiteto de *software* de aplicações do PON poderá usufruí-lo dentro do processo de Desenvolvimento Orientado a Notificações (DON). O DON é um método para projetos de *software* que compreende as fases de requisitos e projeto de um processo de *software* PON (*i.e. Framework* PON para ser preciso). Mais especificamente, quando aplicado ao (já universalizado) processo *Rational Unified Process* (*RUP*), o DON envolve a disciplina de "Requisitos" e adapta a disciplina de "Análise e Projeto" a fim de satisfazer as necessidades de modelagem do PON. A Figura 15 ilustra o *RUP* e a contextualização do DON dentro desse processo [WIECHETECK, 2011].

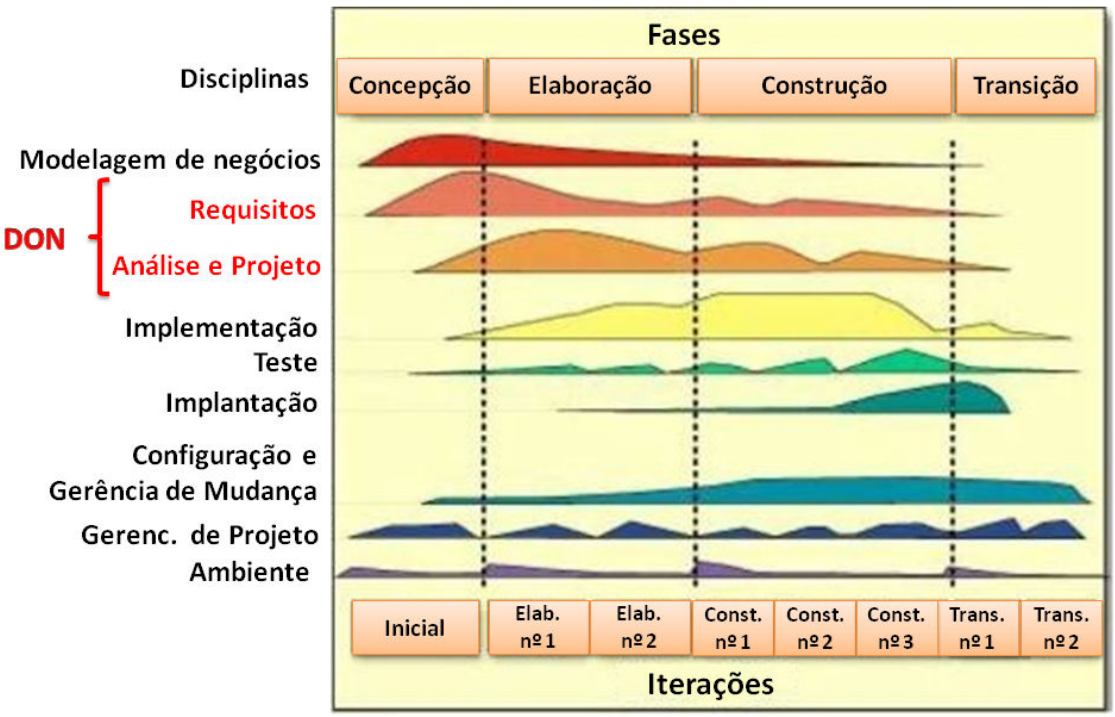


Figura 15 – DON contextualizado no *RUP* [WIECHETECK, 2011]

Aplicando-se nessas disciplinas, o DON apresenta-se como um método iterativo e incremental viabilizando sua adaptação ao *RUP*. O DON é composto de oito etapas que envolvem a criação de diagramas estruturais e comportamentais. O primeiro grupo descreve os elementos estruturais que compõem o sistema,

representando suas partes e seus relacionamentos. Ao seu turno, o segundo grupo descreve o comportamento dos elementos e suas interações [WIECHETECK, 2011].

As oito etapas do método DON são: Capturar Requisitos, Criar Modelo de Casos de Uso, Criar Modelo de Classes, Criar Modelo de Estados de Alto Nível, Criar Modelo de Componentes, Criar Modelo de Sequência, Criar Modelo de Comunicação e Criar Modelo de Redes de Petri. As duas primeiras são referentes à disciplina de "Requisitos" do *RUP*, e as seis restantes são referentes à disciplina de "Análise e Projeto". A Figura 16 exibe o Método DON e suas etapas [WIECHETECK, 2011].

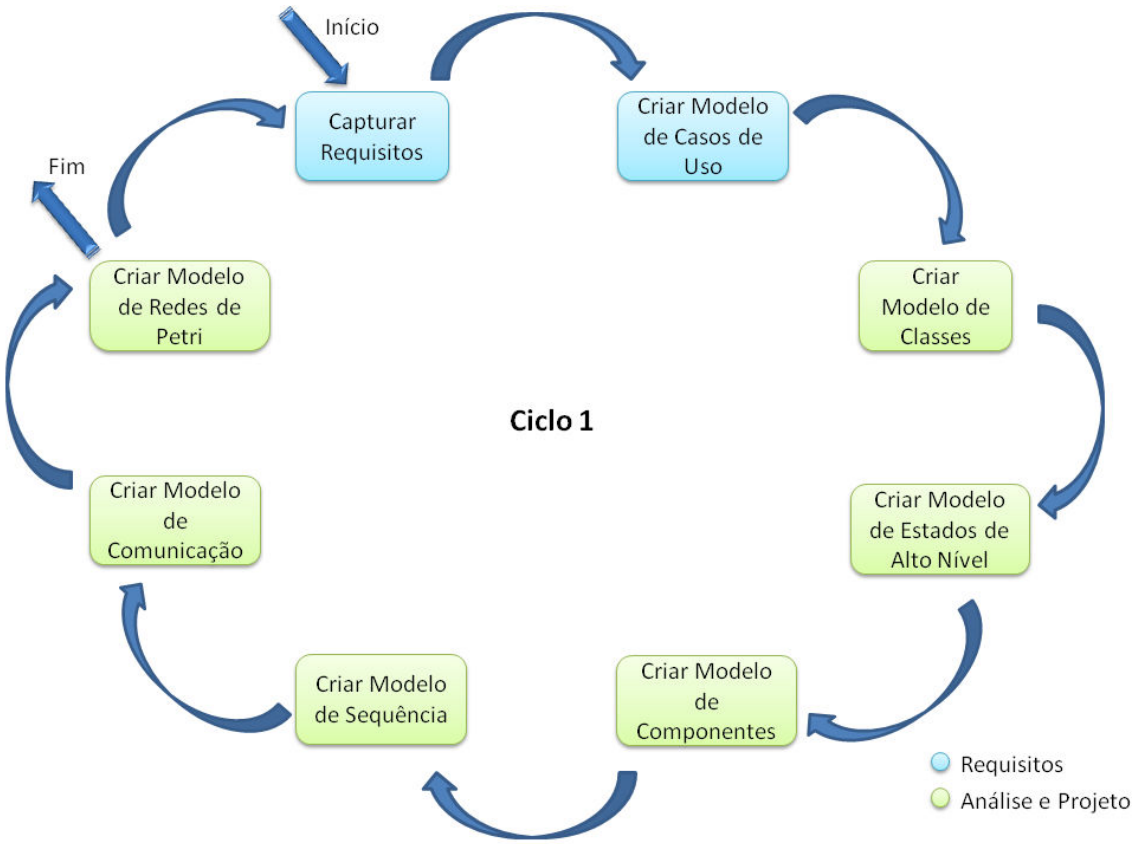


Figura 16 – Método DON [WIECHETECK, 2011]

Fazendo uso do *NOP Profile*, o DON guia os projetistas no desenvolvimento de projetos no PON por meio de etapas. Basicamente o método inicia-se com o levantamento dos requisitos e definição dos casos de uso no Modelo de Casos de Uso. Na sequência o projetista deve criar o Modelo de Estados e Alto Nível a fim de identificar as regras do *software*. Uma vez identificadas, as regras têm sua estrutura estática modelada, de forma inovadora, pelo Modelo de Componentes. Com esta estrutura definida, as regras têm suas iterações modeladas nos Modelos de

Sequência e de Comunicação. Por fim, a modelagem dinâmica das regras é modelada no Modelo de Redes de Petri, obtido a partir do mapeamento do Modelo de Componentes [WIECHETECK, 2011].

2.3.3 Reflexão

Através do método DON com a utilização do *NOP Profile* é possível conceber os artefatos relacionados às fases de levantamento de requisitos e análise/projeto de *software,* as quais compõem as fases de concepção e elaboração do modelo *RUP.* O restante desta dissertação é focada na etapa de implementação e testes, compondo a próxima fase de construção de aplicações. Ainda, este trabalho busca aliar as boas práticas e padrões existentes, bem como a proposta de novas abordagens, algo ainda pouco explorado no âmbito de concepção de *software* no PON.

2.4 APLICAÇÕES UTILIZADAS NA ESTRUTURA DO TRABALHO

Esta seção tem por objetivo descrever os casos de estudo utilizados neste trabalho. Tais casos de estudo foram concebidos inicialmente em C++ sob uma programação orientada a objetos. Posteriormente, esses mesmos sistemas foram reimplementados sob os princípios do PON, com o auxílio de seu *framework*.

Esta seção se apresenta essencial para uma abordagem orientada a exemplos no tocante da apresentação dos demais conteúdos do referencial teórico. Neste sentido, a Subseção 2.4.1 apresenta um simulador com características do clássico jogo *Pacman*. A Subseção 2.4.2, por sua vez, apresenta a implementação de um sistema de pedido de vendas. Por fim, a Subseção 2.4.3 apresenta a implementação de um simulador de voo simples.

2.4.1 Simulador de jogo (*Pacman*)

De modo geral, este caso de estudo consiste na implementação de um simulador com características do clássico jogo *Pacman*[3] [PITTMAN, 2011]. Tal qual o jogo de inspiração, o ambiente possui corredores que formam um labirinto, limitando as ações de movimento dos personagens no cenário, nomeadamente o *Pacman* e seus inimigos, os Fantasmas. Ainda, os personagens do simulador apresentam comportamento autônomo e predeterminadado, ou seja, não são controlados por um usuário [RONSZCKA *et al.*, 2011]. A Figura 17 ilustra o ambiente gerado pelo simulador.

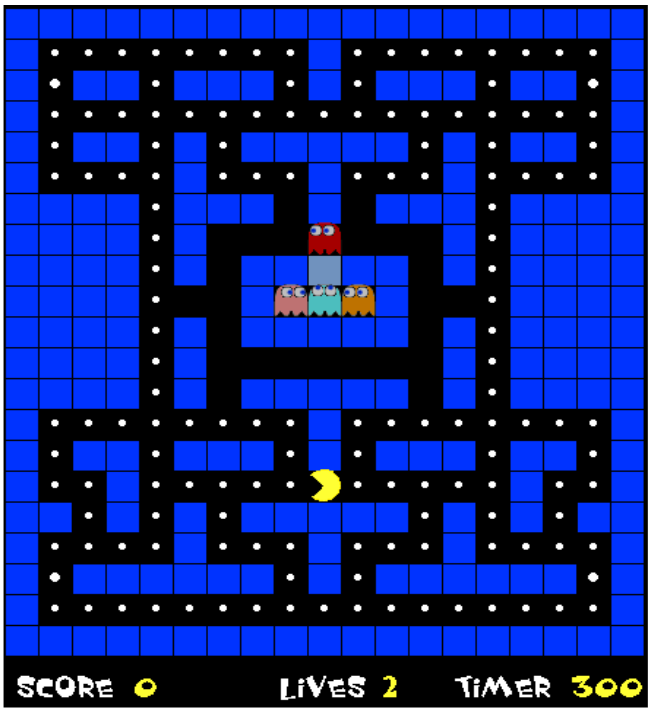


Figura 17 – Ambiente gerado pelo simulador [RONSZCKA *et al.*, 2011]

No início da simulação, o *Pacman* está livre para absorver as pastilhas que se encontram nos corredores e, consequentemente, acumular pontos, procurando evitar contato com os Fantasmas que tentam colidir com ele. O *Pacman* é

[3] Os créditos do jogo e seus direitos autorais pertencem ao indivíduo que o produziu ou a empresa que o publicou. A utilização neste trabalho visa apenas o estudo acadêmico, sem fins lucrativos. Portanto, qualquer outro uso para este conteúdo, poderá estar violando os direitos do autor.

posicionado na posição (9,15) do *grid*, sendo a origem situada no canto superior esquerdo (0,0). Os Fantasmas são posicionados em uma prisão localizada no centro do *grid*, podendo sairem após um determinado tempo.

Para o *Pacman*, o objetivo principal do jogo é maximizar os pontos ganhos. O *Pacman* acumula pontos percorrendo os corredores do labirinto em busca de pastilhas, que são encontradas por todo o labirinto, dispondo de 300 passos para tal. O labirinto é composto por dois tipos de pastilhas; a pastilha normal que quando o *Pacman* a absorve proporciona 10 pontos; e as pastilhas energizadoras proporcionam 50 pontos. Ao absorver uma pastilha energizadora, os papéis do jogo se invertem por um determinado tempo, onde o *Pacman* deixa de ser a presa e passa a ser o predador.

No período de latência, após a absorção de uma pastilha energizadora, os Fantasmas ficam indefesos (assustados) e assumem o papel de presa. Com isso, procuram se distanciar do *Pacman*. Neste período, o *Pacman* pode acumular pontos colidindo com cada um dos Fantasmas assustados. Ao colidir com um Fantasma assustado, ele recebe 200 pontos. A cada colisão adicional no mesmo período de latência, ele acumulará duas vezes mais pontos do que a colisão anterior. Desta forma, as colisões adicionais valerão 400, 800 e 1.600 pontos, respectivamente.

A pontuação máxima possível atingida pelo *Pacman* é de 13.660 pontos. O *Pacman* tem um determinado tempo para acumular o maior número de pontos possíveis, dispondo de três vidas para tal. Fora do período de latência, cada colisão entre o *Pacman* e um Fantasma, resulta na perda de uma de suas vidas e no reposicionamento dos personagens em suas posições iniciais no *grid*.

Particularmente, neste simulador, os personagens possuem um campo visual que representa a profundidade com que eles enxergam os corredores do labirinto. A Figura 18 ilustra um exemplo de campo visual com profundidade máxima, de valor 5. A visão dos personagens abrange todas as direções, sendo que o alcance de visão desses é representado por círculos, limitando-se ao se deparar com o fim de um corredor.

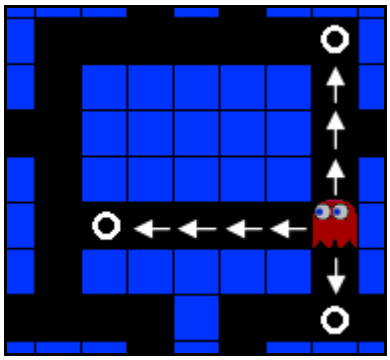


Figura 18 – Exemplo de campo visual com profundidade 5

O simulador apresenta particularidades que beneficiam a implementação de regras para a movimentação dos personagens nos corredores do labirinto. Dentre tais particularidades, tem-se a classificação de esquinas em categorias. As esquinas representam o encontro ou cruzamento de dois ou mais corredores que compõem o labirinto, cada qual com seu formato distinto.

O labirinto é formado por 9 diferentes formatos de esquinas, com o intuito de minimizar a quantidade de regras, visto que o tratamento das ações dos personagens se baseia nessa classificação e não em todas as partes do labirinto. Conforme ilustra a Figura 19, cada formato particular de esquina é representado por um valor, presente em uma escala de 1 a 9.

Figura 19 – Labirinto dividido em 9 diferentes tipos de esquinas

Conforme apresentado na Figura 18, os personagens possuem determinada capacidade de visão que lhes permite perceber os elementos que compõem o labirinto. Por exemplo, o *Pacman* enxerga pastilhas, paredes, Fantasmas e corredores vazios. Similarmente, os Fantasmas enxergam paredes, corredores vazios, o *Pacman* e os demais Fantasmas. Desta forma, baseado em sua visão, os personagens devem tomar decisões coerentes para se moverem pelos corredores do labirinto em busca de alcançar seus objetivos.

As ações de movimentação adotadas pelos personagens estão ligadas aos elementos percebidos em seus campos visuais, por meio dos corredores do

labirinto. Para cada um dos personagens, eventos são disparados à medida que esses detectam elementos em seus campos visuais. Normalmente, mais de um elemento é detectado a cada vez que um personagem se encontra em uma esquina. De forma a evitar conflitos, somente um evento pode ser disparado a cada momento, sendo necessário definir uma escala de prioridades para tal. Os eventos que incitam as regras que movimentam o *Pacman* apresentam a seguinte ordem de prioridade [RONSZCKA *et al.*, 2011]:

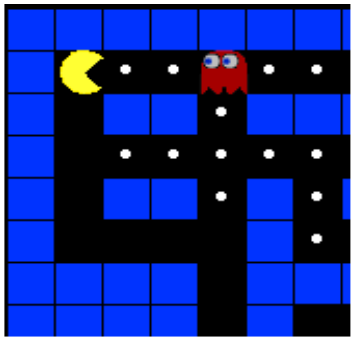
- **Fantasma em estado normal detectado**: é o evento com a maior prioridade; ele é incitado quando o *Pacman* avistar um Fantasma em estado normal em seu campo visual;
- **Fantasma em estado assustado detectado**: é o evento com a segunda maior prioridade; ele é incitado no período de latência, após a absorção de uma pastilha energizadora, quando o *Pacman* avistar um Fantasma em estado assustado em seu campo visual;
- **Pastilha detectada**: este é o evento com a terceira maior prioridade; ele é incitado quando o *Pacman* avistar uma pastilha normal ou uma pastilha energizadora em seu campo visual;
- **Corredor vazio**: este é o evento com a menor prioridade; ele é incitado quando nenhum dos outros eventos tiver sido instigado.

Por sua vez, os eventos que incitam as regras que movimentam os Fantasmas apresentam a seguinte ordem de prioridade:

- ***Pacman* detectado enquanto o Fantasma estiver em estado assustado**: é o evento com a maior prioridade; ele é incitado quando o Fantasma, em estado assustado, avistar o *Pacman* em seu campo visual;
- ***Pacman* detectado enquanto o Fantasma estiver em estado normal**: é o evento com a segunda maior prioridade; ele é incitado quando o Fantasma, em estado normal, avistar o *Pacman* em seu campo visual;

- **Outro Fantasma detectado**: é o evento com a terceira maior prioridade; ele é incitado quando o Fantasma avistar outro Fantasma no mesmo corredor em seu campo visual.
- **Corredor vazio**: este é o evento com a menor prioridade; ele é incitado quando nenhum dos outros eventos tiver sido instigado.

Os personagens só alteram sua direção quando estiverem sobre uma esquina, podendo retornar ao caminho pelo qual vieram ou escolher outro corredor para seguir. A Figura 20 ilustra exemplos de regras de movimentação para o personagem *Pacman*.



Se:
esquina atual **=** 1 [*DownRight*] **e**
evento detectado **=** fantasma [direita] **e**
Então:
"Fugir do fantasma movendo-se para baixo"
Se:
esquina atual **=** 1 [*DownRight*] **e**
evento detectado **=** pastilha [direita] **e**
Então:
"Absorver pastilha avançando para direita"

Figura 20 – Exemplos de regras de movimentação do *Pacman*

De fato, as regras que movimentam os personagens nos corredores do labirinto são baseadas nas percepções desses. Tais percepções representam a esquina atual, os eventos detectados ordenados por prioridade e a localização (direção) do elemento prioritário detectado (*i.e.* no caso da Figura 20, a primeira regra é composta pelo evento de maior prioridade, disparada do corredor à direita) [RONSZCKA *et al.*, 2011].

Ademais, as regras que movimentam os personagens em cada situação particular são predefinidas no simulador. Ainda, de modo a facilitar o entendimento do fluxo principal de execução dessa aplicação, o diagrama de atividades ilustrado na Figura 22 esboça o núcleo de sua execução.

Devido às dimensões do diagrama original, o Diagrama de Classes ilustrado na Figura 21 abstrai a complexidade do simulador, externando a essência do mesmo por meio de suas classes principais.

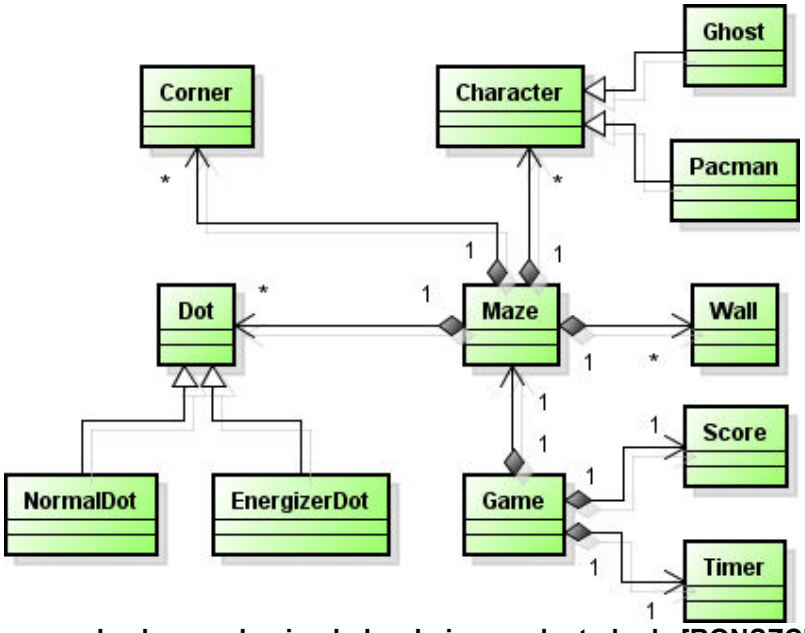


Figura 21 – Diagrama de classes do simulador do jogo, adaptado de [RONSZCKA *et al.*, 2011]

A classe *Game* comporta a estrutura de funcionamento do simulador, composta pelas classes *Maze*, *Score* e *Timer*. A classe *Maze* representa a estrutura do Labirinto, formado por Paredes (*Walls*), Esquinas (*Corners*), Pastilhas (*Dots*) e Personagens (*Characters*). O cerne do simulador, por sua vez, é constituído pelas regras que movimentam os personagens nos corredores do labirinto, concentradas nas classes *Ghost* e *Pacman*.

Ainda, de modo a facilitar o entendimento do fluxo principal de execução dessa aplicação, o diagrama de atividades ilustrado na Figura 22 esboça o núcleo de sua execução.

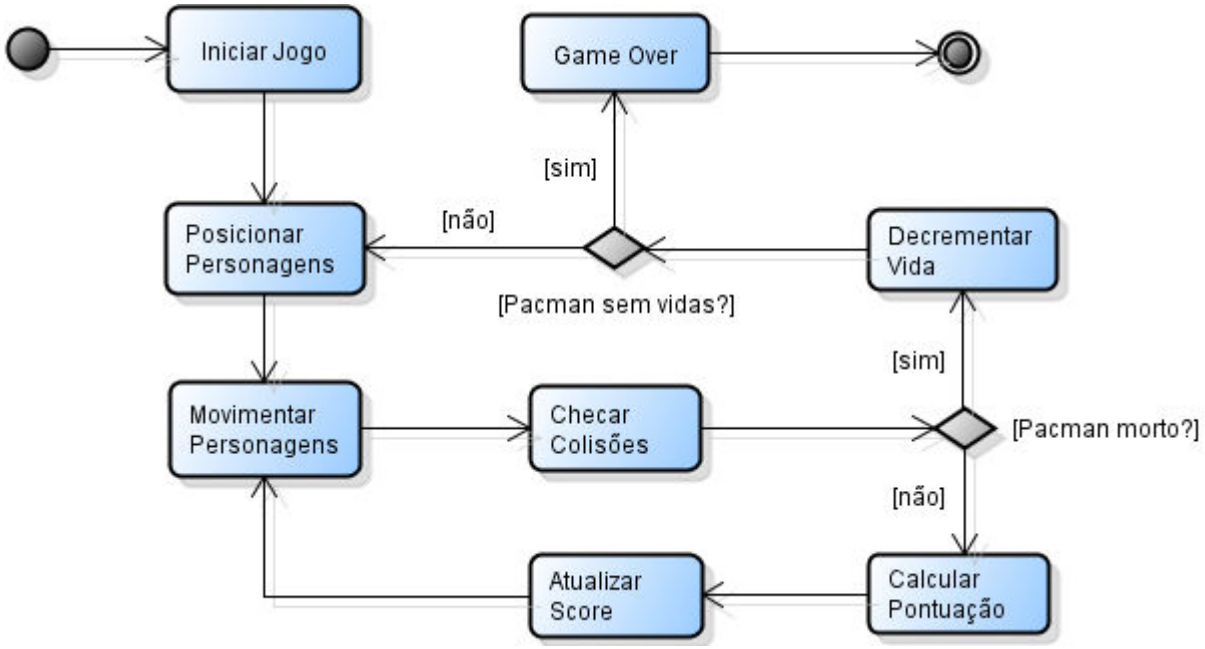


Figura 22 – Diagrama de atividades do simulador de jogo *Pacman*

Conforme apresentado no diagrama de atividades ilustrado na Figura 22, o fluxo de execução da aplicação ocorre em torno da movimentação dos personagens no labirinto. Ao passo que os personagens se movimentam, checagens de colisão e subsequentes tratamentos são realizados de modo a definir o fluxo de execução do jogo.

2.4.2 Sistema de pedido de vendas

De modo geral, este caso de estudo consiste na implementação de um sistema de pedido de vendas usual. Essa aplicação consiste em uma tradicional aplicação *CRUD* (acrônimo de *Create, Retrieve, Update* e *Delete*). Em suma, o escopo dessa aplicação é composto pelo diagrama de casos de uso ilustrado na Figura 23.

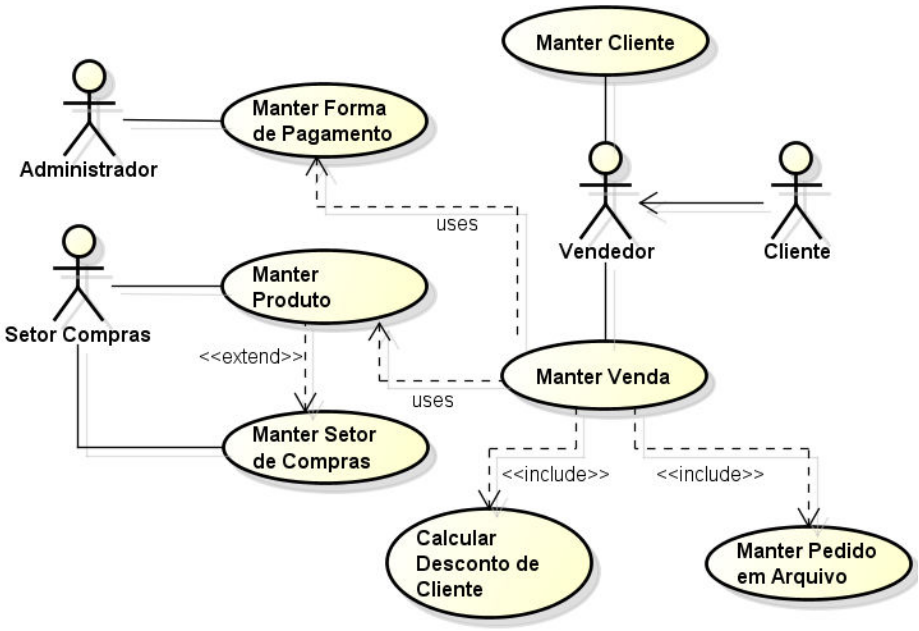


Figura 23 – Casos de uso do sistema de pedido de vendas

Conforme ilustra a Figura 23, o ator Administrador é responsável por manter as informações do cadastro de formas de pagamentos. O ator Setor de Compras, por sua vez, é responsável por cadastrar e atualizar as informações de produtos e do próprio setor de compras. Ainda, o ator Cliente, solicita uma venda a um respectivo ator Vendedor. Este, por sua vez, cadastra o cliente e efetua a venda propriamente dita.

Para elucidar o comportamento e as responsabilidades de cada caso de uso da Figura 23, a Figura 24 apresenta um diagrama de atividades da execução de um pedido de venda.

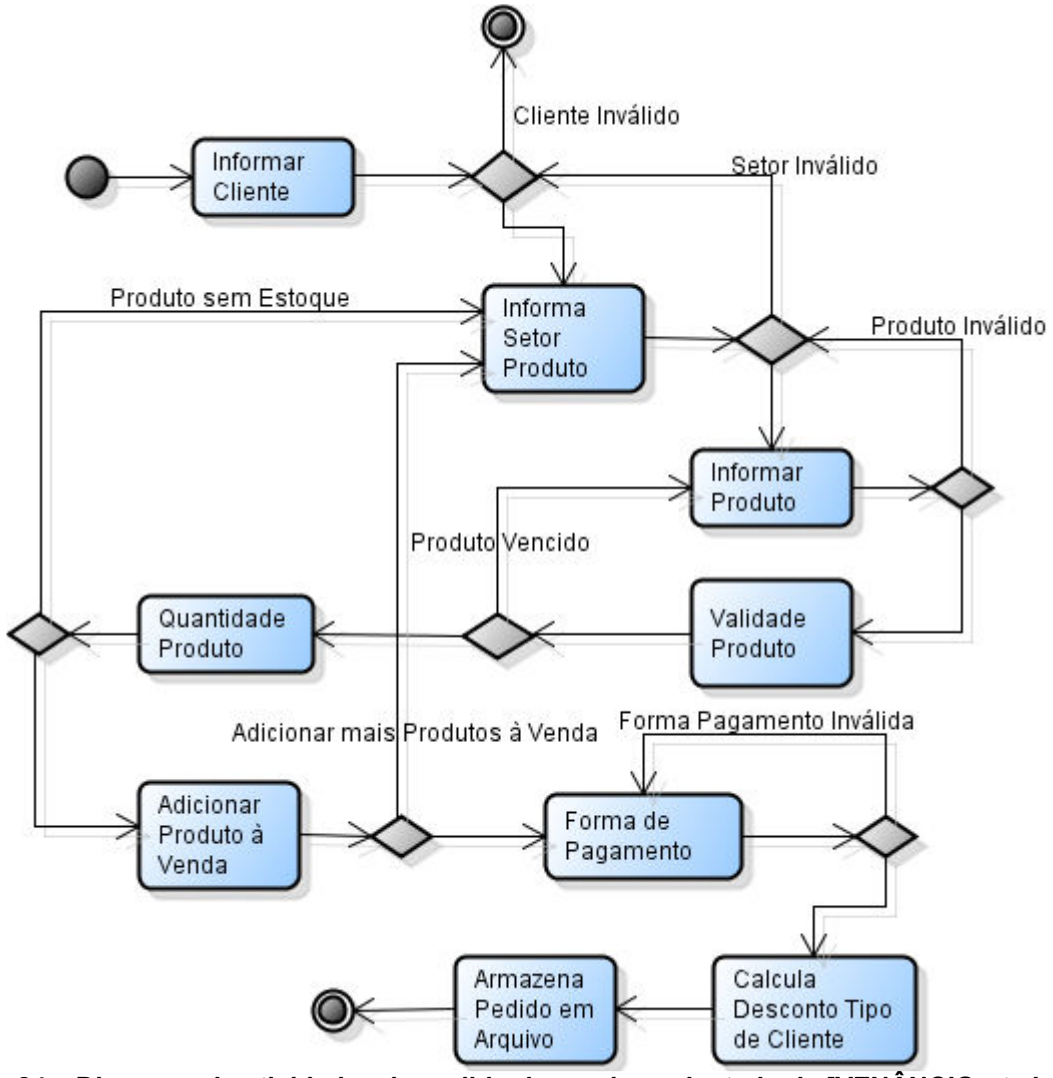


Figura 24 – Diagrama de atividades do pedido de vendas, adaptado de [VENÂNCIO *et al.,* 2011]

Conforme ilustrado na Figura 24, inicialmente o cliente (denotado pelo ator cliente) solicita a venda para um respectivo vendedor (denotado pelo ator vendedor). Assim o vendedor informará o respectivo cliente que realizará o pedido. Uma vez escolhido e aprovado a venda para determinado cliente, devem ser informados os produtos que irão compor o pedido. O sistema possui validações quanto à existência de produtos e clientes. Ademais, verifica-se o estoque disponível de tais produtos. Se o produto escolhido para venda pertencer ao setor de perecíveis, verifica-se ainda a sua data de validade. Na ocorrência de produtos vencidos, a venda não será permitida [VENÂNCIO *et al.*, 2011].

Após todo o ciclo de informe de produtos, a venda poderá ser finalizada após a inserção da forma de pagamento. Na implementação desse sistema, existem apenas duas formas de pagamento possíveis, à Vista ou à Prazo. O cliente, em seu cadastro, possui uma informação sobre seu limite de crédito. Caso a forma de

pagamento escolhida tenha sido à Prazo, o sistema verifica se o cliente tem permissão para efetuar a compra, confrontando o valor total do pedido com seu limite de crédito. Ademais, no cadastro do cliente há uma informação que lhe concede um tipo de classificação. Utiliza-se tal classificação para a concessão de descontos especiais durante a finalização da venda. Para tanto, existe um total de 20 tipos de classificação de clientes que dispõem de descontos que variam de uma faixa de 5% a 95% [VENÂNCIO *et al.*, 2011].

De modo a facilitar o entendimento do escopo desse sistema, o diagrama de classes ilustrado na Figura 25 externa a essência da implementação do sistema de Vendas por meio de suas principais classes.

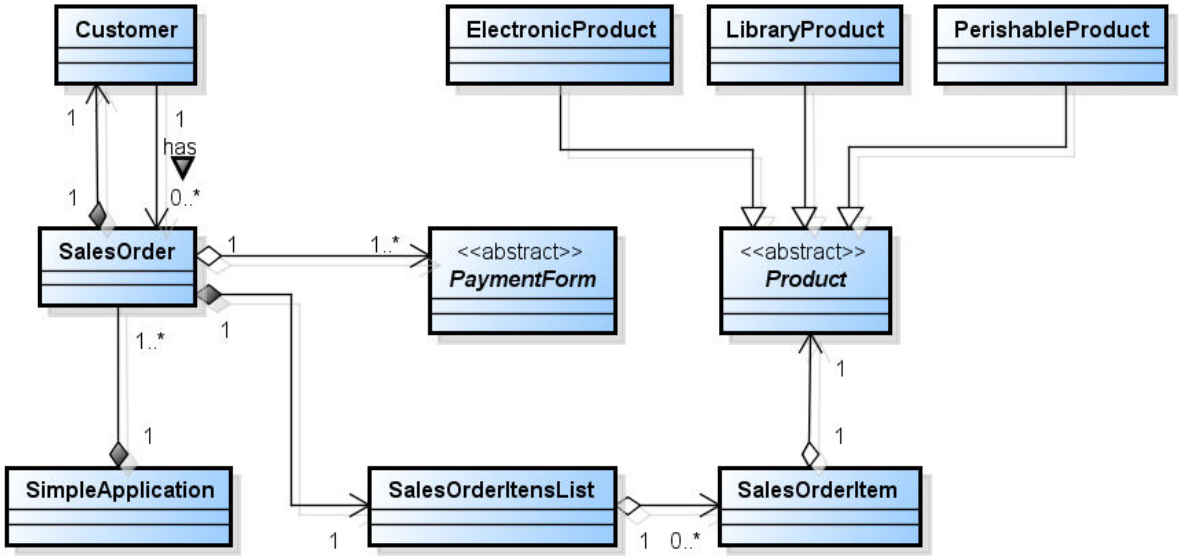


Figura 25 – Diagrama de classes do sistema de vendas, adaptado de [VENÂNCIO *et al.*, 2011]

Conforme ilustrado na Figura 25, a classe *SalesOrder* possui a responsabilidade de armazenar os dados de um pedido. Basicamente, um pedido é composto por um cliente, uma forma de pagamento e uma lista de itens do pedido. Em tal lista são armazenados os produtos e suas respectivas quantidades. A classe *Product* é definida como abstrata, podendo ser estendida para quaisquer tipos de produtos que possam a vir a ser implementados. Ainda a classe principal denominada *SimpleApplication*, define a instanciação dos *FBEs* e as suas respectivas *Rules.* Ademais, a execução da aplicação propriamente dita ocorre através da execução do método *codeApplication* da classe *SimpleApplication.*

A título de exemplificação, a Figura 26 demonstra a composição da *Rule* responsável por finalizar uma venda. Nela estão relacionadas as *Premises* que

deverão ser satisfeitas para que a finalização da venda ocorra. Assim, a primeira *Premise* verificaria se a forma de pagamento selecionada foi a prazo, neste caso é necessário validar o limite de crédito disponível para o cliente, o qual faria parte da segunda *Premise* da *Rule* em questão. A terceira e última *Premise* validaria o tipo de desconto concebido para o cliente em questão, o qual possui 20 possíveis tipos de descontos.

Se:			
Forma Pagamento	**=**	À Prazo	**e**
Limite Crédito Cliente	**>=**	Total da Venda	**e**
Tipo Desconto Cliente	**=**	1	
Então:			
Conceder Desconto do tipo 1 (um)			
Finalizar Venda			

Figura 26 – Exemplo de regra "finalizar vendas"

Conforme descrito no paragrafo anterior, existem 20 tipos de descontos que poderão ser concedidos aos clientes. Neste caso para cada tipo de desconto será criado uma *Rule* correspondente, conforme apresentada na Figura 26. Desta forma, após a satisfação das *Premises* supracitadas, a *Rule* em questão concederia o desconto do pedido de vendas para o cliente e por fim finalizaria sua venda. Neste caso o método denominado finalizar venda gravaria os dados em arquivo.

Ainda a *Rule* esboçada pela Figura 27 validaria a inclusão de um determinado produto aos itens de compra do respectivo cliente. Assim, a cada *FBE Product* inserido, a *Rule* em questão verificaria se a data de validade do produto é maior que a data atual e se o estoque atual do produto é maior ou igual a quantidade solicitada pelo cliente. Após a satisfação das *Premises* supracitadas, o método responsável por adicionar o produto à lista de produtos seria invocado.

Se:			
Validade Produto	**>**	Data Atual	**e**
Estoque Produto	**>=**	Quantidade Vendida	
Então:			
Adicionar Produto			

Figura 27 – Exemplo de uma regra "adicionar produto"

2.4.3 Simulador de voo

De modo geral, este caso de estudo consiste na implementação de um simulador para o controle de aeronaves. Por definição, uma aeronave é qualquer máquina capaz de sustentar voo, sendo em sua grande maioria capaz de alçar voo por meios próprios. Com o objetivo de facilitar o entendimento da implementação do caso de estudo em questão, a Figura 28 ilustra uma descrição simplificada das partes componentes de uma aeronave.

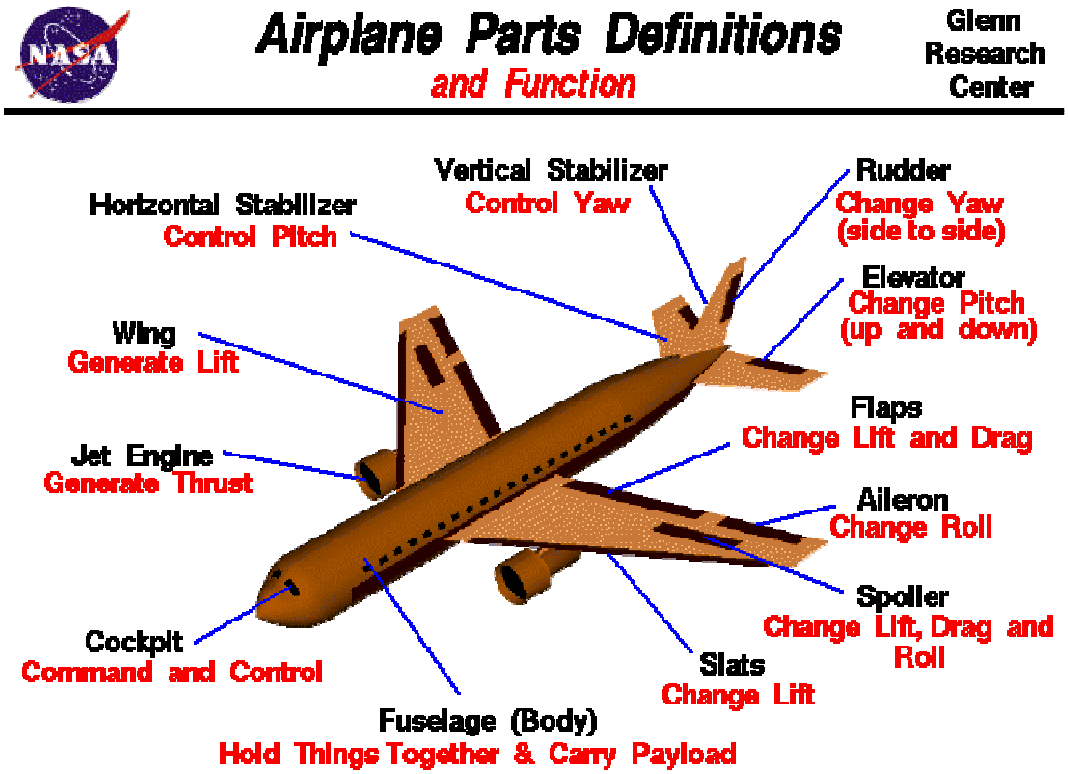


Figura 28 – Descrição simplificada das partes componentes de uma aeronave [NASA, 2012]

Sucintamente, para qualquer aeronave voar, o peso dessa precisa ser levantado, incluindo o peso dos passageiros, cargas, combustível etc. As asas geram a maior parte da elevação e sustentação necessária para manter um avião no ar. Para gerar a elevação (*lift*) da aeronave e sua sustentação subsequente, essa deve ser empurrada através do ar. O ar resiste ao movimento sob a forma aerodinâmica de arrasto (*drag*). Os motores da turbina, que estão localizados sob as asas, fornecem o impulso (*thrust*) necessário para superar tal resistência gerada pelo ar [NASA, 2012].

Outrossim, as asas possuem uma parte móvel denominada *aileron*, com a finalidade de rolar (*roll*) o avião sob o eixo longitudinal (eixo x). Ainda, as asas possuem componentes adicionais, denominados *flaps*, *slats* e *spoilers*. Os *flaps* e *slats* possuem basicamente funções para proporcionar força para a asa e estabilizar o voo durante a decolagem e aterrissagem do avião. Os *spoilers* também são utilizados durante a aterrissagem para frenar o avião e para reagir aos *flaps* quando a aeronave estiver no solo [NASA, 2012].

Ademais, para controlar e manobrar a aeronave, asas menores estão localizadas na parte traseira do avião. A cauda de uma aeronave possui um formato específico para manter o voo estável, através da presença de partes fixas, denominadas estabilizadores horizontais e vertical. O estabilizador vertical apresenta anexo a si um leme (*rudder*), que é uma parte móvel com a finalidade de controlar a guinada (*yaw*) de uma aeronave, movendo a direção do avião para esquerda ou para direita (eixo z). Os estabilizadores horizontais, por sua vez, apresentam partes móveis denominadas elevadores (*elevators*) anexos a eles, os quais possuem a finalidade de controlar a arfagem (*pitch*) de uma aeronave, movendo-a para cima ou para baixo (eixo y) [NASA, 2012].

De maneira geral, a concepção deste caso de estudo se deu por meio da implementação das partes componentes de uma aeronave e suas respectivas funcionalidades. Para um melhor entendimento da estrutura desse projeto, a Figura 29 ilustra o diagrama de classes do cerne do simulador.

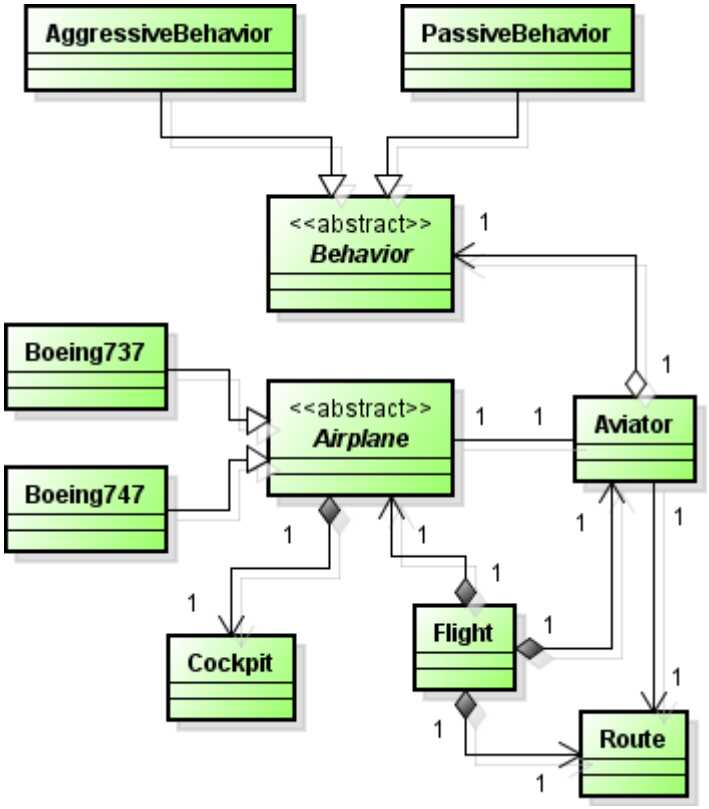


Figura 29 – Representação do cerne do controle de aeronaves

De acordo com a Figura 29, o cerne desse sistema gira em torno de uma simulação de voo (*Flight*), que é constituído por um avião (*Airplane*), um piloto (*Aviator*) e uma rota (*Route*). Basicamente, uma rota é composta por sua origem e destino, definindo restrições para com a forma com que o piloto conduzirá tal voo. O relacionamento entre o piloto e o avião, particularmente, implica na maior parte das chamadas de métodos dessa aplicação.

Ademais, os comandos gerais de um avião estão presentes em sua cabine de pilotagem (*Cockpit*), sendo essa utilizada indiretamente pelo piloto encarregado a comandar tal avião. A cabine de pilotagem de um avião inclui uma série de instrumentos de voo, os quais permitem com que o piloto possa acelerar, guiar, levantar voo e aterrissar a aeronave durante o percurso preestabelecido em seu plano de voo. O diagrama de classes ilustrado na Figura 30 apresenta a estrutura do *Cockpit* e seus respectivos relacionamentos.

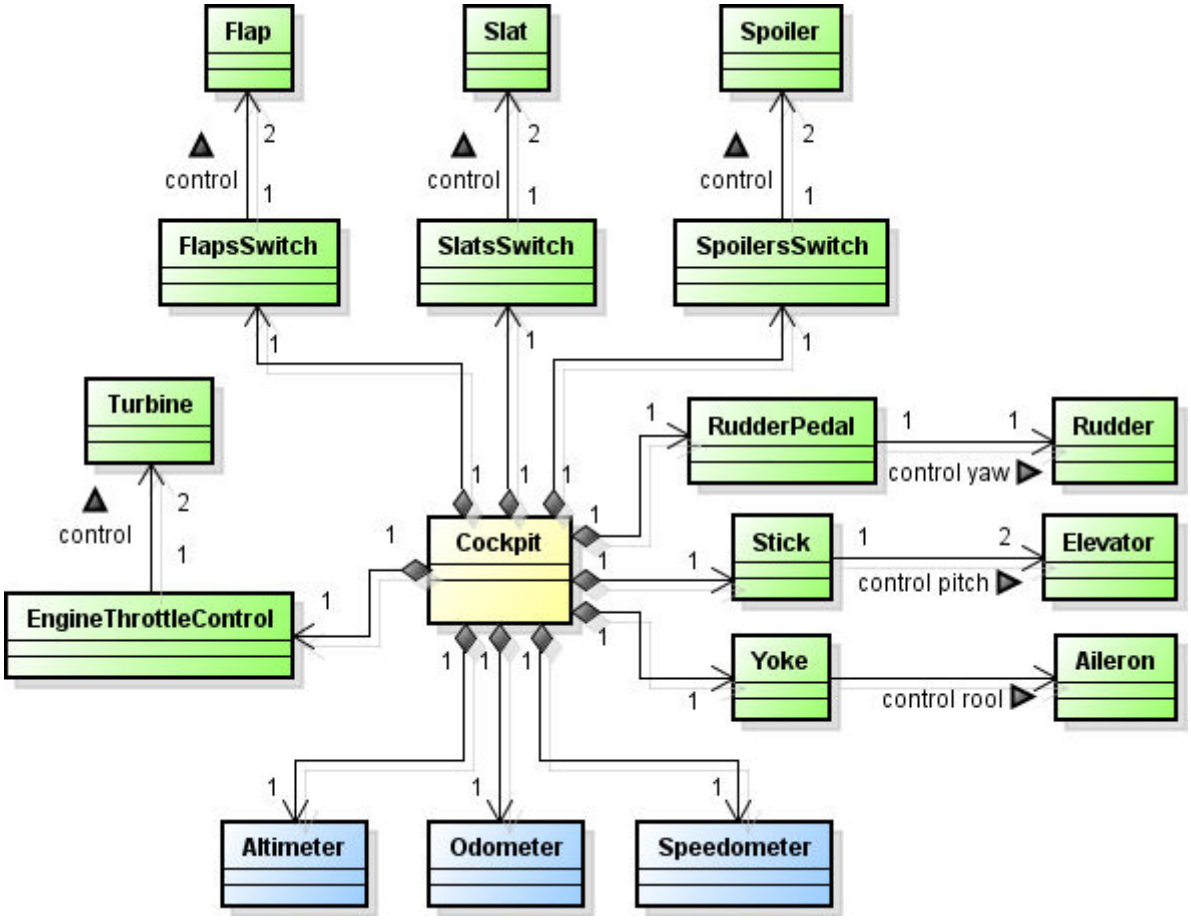


Figura 30 – Estrutura do *Cockpit*

Conforme apresenta a Figura 30, a classe *Cockpit* é composta por outras classes, as quais fornecem uma interface de controle para as demais partes de uma aeronave. As classes altímetro (*Altimeter*), odômetro (*Odometer*) e velocímetro (*Speedometer*) além de controlarem respectivamente a altitude, distância percorrida e velocidade de uma aeronave, atuam com caráter informativo, representando as medidas escalares pertinentes à execução do voo.

As demais classes proporcionam ao piloto o efetivo controle da aeronave, que são: o controlador de aceleração (*EngineThrottleControl*) das turbinas (*Turbine*), a alavanca (*FlapsSwitch*) que controla os *flaps* (*Flap*), a alavanca (*SlatsSwitch*) que controla os *slats* (*Slat*) , a alavanca (*SpoilersSwitch*) que controla os *spoilers* (*Spoiler*), o pedal (*RudderPedal*) que controla o leme (*Rudder*), a alavanca (*Stick*) que controla os estabilizadores horizontais (*Elevator*) e manche (*Yoke*) que controla os ailerons (*Aileron*).

Ainda, a Figura 31 ilustra o diagrama de classes que representa as demais partes componentes de uma aeronave.

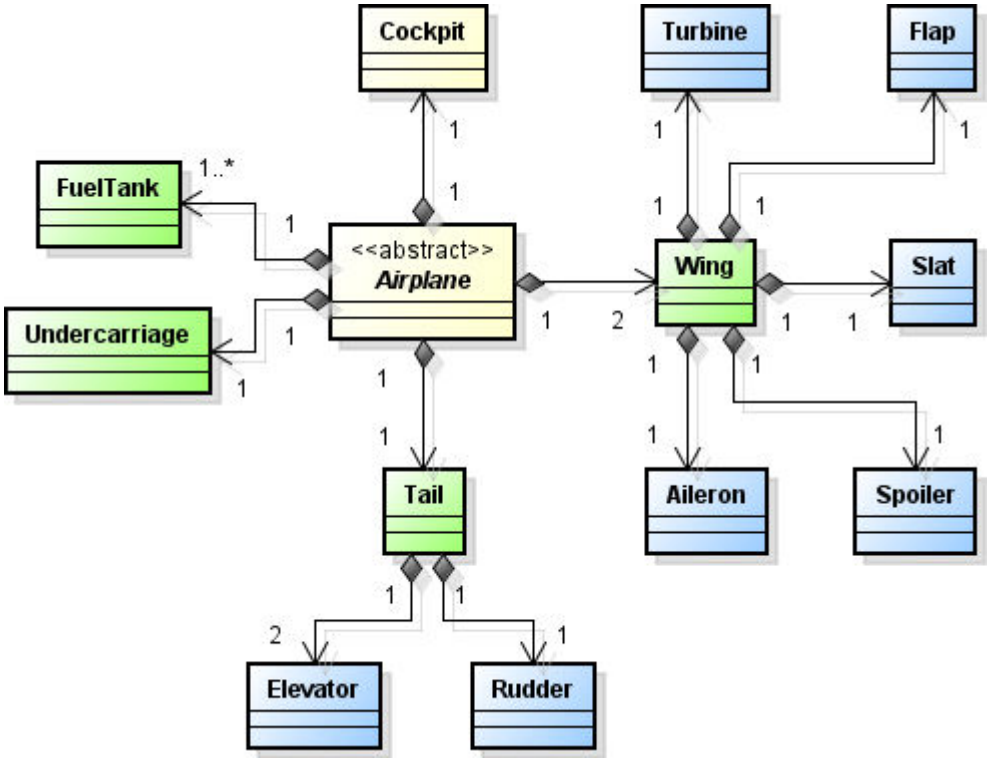


Figura 31 – Estrutura das asas e da calda do avião

Além da classe *Cockpit*, uma aeronave é composta por asas (*Wings*) e demais componentes que formam suas estruturas, calda (*Tail*) e demais componentes que formam sua estrutura, tanque de combustível (*FuelTank*) e trem de pouco (*Undercarriage*).

Ainda, de modo a facilitar o entendimento do fluxo principal de execução dessa aplicação, o diagrama de atividades ilustrado na Figura 32 esboça o núcleo de sua execução.

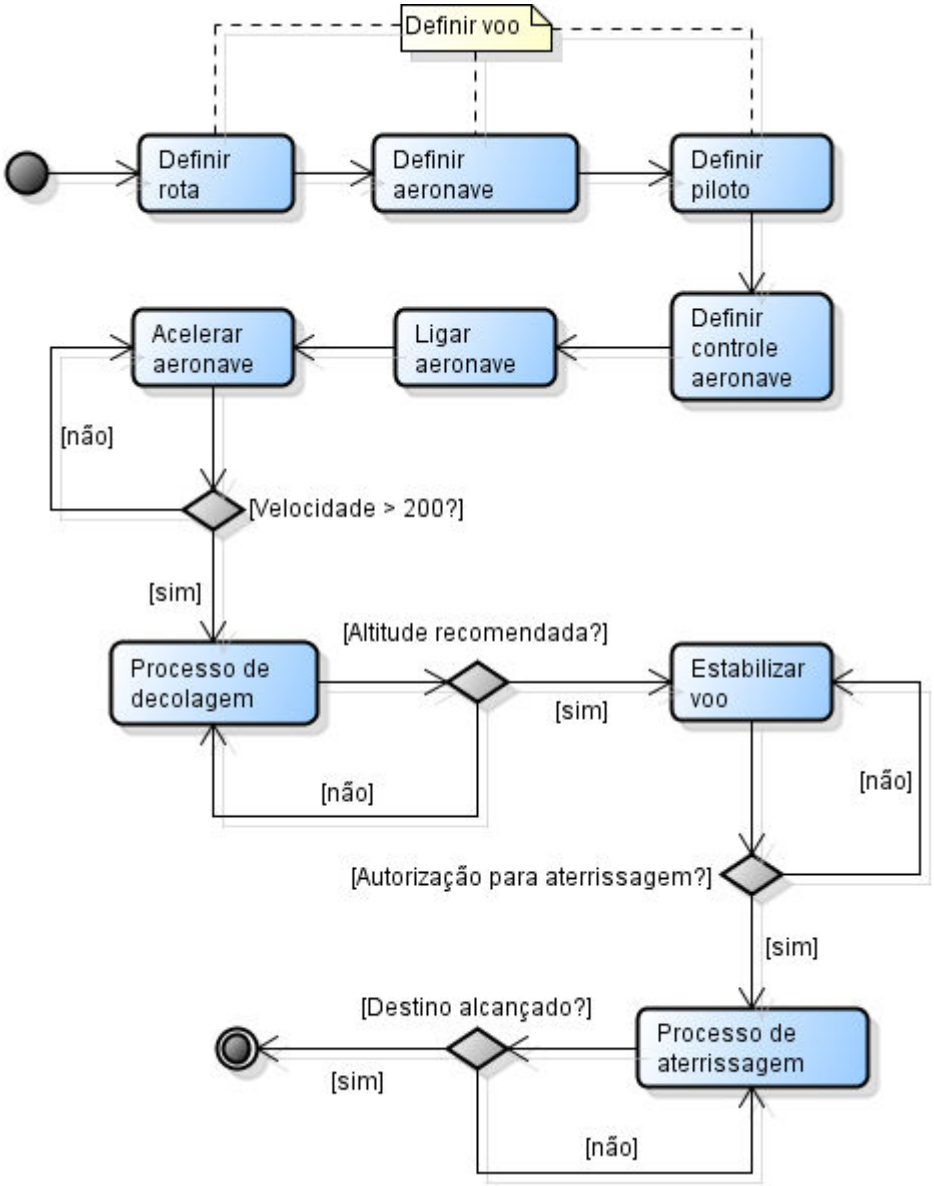


Figura 32 – Diagrama de atividades do simulador de voo

Conforme apresentado no diagrama de atividades ilustrado na Figura 32, o fluxo de execução da aplicação gira em torno do controle da aeronave, desde o processo de decolagem, estabilidade do voo, até o processo de aterrissagem. Em tais processos, todas as classes de controle apresentadas estão envolvidas.

2.5 PARTICULARIDADES DE DESENVOLVIMENTO DE UM PROJETO DE *SW*

De maneira geral, o desenvolvimento de um projeto de *software* apresenta particularidades que podem tornar sua concepção simples ou complexa, dependendo das técnicas e metodologias aplicadas.

Neste âmbito, padronizações na estrutura geral de um *software* auxiliam, de certa forma, a sua concepção, uma vez que proveem diretrizes aos projetistas para que minimizem o tempo gasto na busca por soluções de problemas que se apresentam comuns nesse processo.

Outrossim, em um nível de abstração maior no processo de desenvolvimento de um projeto de *software* estão os padrões arquiteturais. Esses definem um esquema para a organização estrutural de projetos de *software*. Tais padrões provem um conjunto de subsistemas predefinidos, especificam suas responsabilidades, e incluem regras e diretrizes para organizar os relacionamentos entre eles [BUSCHMANN *et al.*, 1996].

Ademais, um padrão arquitetural é normalmente composto por padrões de projeto, os quais proveem um esquema adaptável para o refinamento da estrutura interna, ou até mesmo, do relacionamento de tais subsistemas. Ademais, um padrão de projeto descreve uma estrutura comumente recorrente de comunicação entre componentes de *software* menores, que resolve de uma maneira particular um problema de projeto dentro de um contexto específico [BUSCHMANN *et al.*, 1996].

Ainda, em um nível mais baixo aos dos padrões de projeto se encaixam os padrões de implementação ou de codificação. Esses são conjuntos de boas práticas que normalmente definem a maneira com que o *software* deve ser escrito. Ademais, os padrões de implementação definem regras para a composição da estrutura de módulos menores de uma aplicação, tais como a forma de concepção de atributos e métodos de uma classe, por exemplo.

Neste sentido, esta seção tem por objetivo apresentar algumas técnicas e boas práticas para a concepção de projetos, de maneira a amenizar ou, até mesmo, eliminar alguns dos principais sintomas de um projeto mal estruturado. Para isso, a Subseção 2.5.1 apresenta tais sintomas, enquanto as subseções 2.5.2 e 2.5.3 contextualizam algumas boas práticas de programação, bem como os princípios de projeto mais relevantes para a concepção de *software*.

2.5.1 Sintomas de um projeto mal estruturado

A criação de projetos de *software* normalmente demanda um esforço significativo para a sua concepção. Nesse processo, além de possíveis mudanças ao longo do desenvolvimento, o projeto quando mal estruturado, tende a apresentar alguns sintomas de forte acoplamento e baixa coesão que podem ocasionar problemas futuros em sua essência como um todo.

Em [MARTIN e MARTIN, 2006] são descritos alguns dos sintomas que indicam que um projeto apresenta sintomas de má estruturação, que são:

- **Rigidez**: é a tendência de o *software* ser difícil de modificar, incluindo pequenas e simples alterações. Um projeto é considerado rígido quando uma única alteração causa uma cascata de alterações subsequentes em módulos dependentes.
- **Fragilidade**: estreitamente relacionada à rigidez; é a tendência de uma única alteração afetar o comportamento de muitas funcionalidades distintas de um *software*. Frequentemente, novos problemas aparecem em partes não relacionadas conceitualmente com o trecho de código alterado.
- **Imobilidade**: é a inabilidade de reusar trechos de código de outros projetos ou de um mesmo projeto na composição de novas entidades computacionais. Um projeto é considerado imóvel quando contém partes que podem ser úteis em outros sistemas, mas o esforço e o risco envolvido para separar tais partes do sistema original inviabilizam a sua realização.
- **Viscosidade**: representa a dificuldade em se tomar a decisão certa perante uma alteração no *software*. Geralmente, os desenvolvedores encontram mais de uma maneira de realizar determinada alteração no código, nas quais, algumas preservam o projeto, enquanto outras não. Quando preservar o projeto envolve maiores dificuldades de implementação, o projeto é considerado de alta viscosidade.

A presença desses sintomas indica que a arquitetura do *software* está má estruturada, desprovendo o *software* dos maiores benefícios presentes na programação orientada a objetos, por exemplo, que são flexibilidade, reusabilidade, e manutenibilidade. Além desses sintomas, outros tendem a aparecer em projetos que mudam com certa frequência [MARTIN e MARTIN, 2006], que são:

- **Complexidade desnecessária**: representa um projeto que contém elementos que não são necessários em um primeiro momento. Isso acontece quando desenvolvedores antecipam a necessidade de alterações futuras, criando um nível de complexidade, muitas vezes, desnecessário.
- **Repetições inúteis**: geralmente ocasionadas pela má reutilização de trechos de código existentes. Usualmente, desenvolvedores despreocupados com a qualidade do *software*, tendem a alterar o código copiado apenas para atender as necessidades imediatas. Esse problema tende a crescer de tal forma que pequenas alterações tendem a gerar inúmeras horas de trabalho, uma vez que cada repetição de código tende a ser minimamente diferente das demais, necessitando correções distintas.
- **Opacidade**: representa a tendência de um módulo ser difícil de entender devido à quantidade de expressões desorganizadas. O código pode ser escrito de uma maneira clara e expressiva ou ele pode ser escrito de uma maneira opaca e complexa.

2.5.2 Boas práticas de programação

A chave para a construção de bons programas está presente na antecipação de novos requisitos e mudanças para os já existentes. Com isso em mente, é possível projetar tais programas de maneira que esses possam evoluir adequadamente a quaisquer mudanças que realmente se façam necessárias [GAMMA *et al.*, 1995]. De maneira a evitar que mudanças na estrutura do *software* ocasionem os sintomas de um projeto mal estruturado (Subseção 2.4.1), os

desenvolvedores podem se atentar as boas práticas de programação de maneira a tornar a estrutura de seus projetos mais flexível e suscetível a mudanças.

Basicamente, a capacidade de manutenibilidade e extensibilidade de um *software* pode ser quantificada pelos atributos de desacoplamento e coesão [MARTIN e MARTIN, 2006].

O acoplamento entre classes é uma medida de interconexão entre elas. Acoplamento forte significa que as classes relacionadas precisam conhecer detalhes internos umas das outras, e que alterações em uma entidade propagam alterações em outras partes do sistema, o que potencialmente dificulta a manutenibilidade e extensibilidade do projeto.

A coesão, por sua vez, mede o grau de conectividade entre os elementos de um único módulo, tanto no seu modelo quanto nas suas instanciações, o que seria respectivamente na sua classe e seus objetos em termos de Orientação a Objetos (OO). Quanto maior a coesão do *software*, melhor definidas e relacionadas estão as responsabilidades de cada classe individual da aplicação. Cada classe tem um conjunto muito específico de ações intimamente relacionadas a serem executadas. Portanto, *softwares* com baixa coesão usualmente apresentam mais dificuldades para manter e estender.

De maneira a atingir um cenário favorável à extensibilidade e manutenibilidade, isto é, um projeto altamente coeso e fracamente acoplado, algumas boas práticas na elaboração de tais projetos podem ser adotadas.

Dentre tais boas práticas, cita-se a correta utilização do encapsulamento. Para isso, duplicações de código devem ser evitadas ao máximo, a fim de evitar um dos pesadelos de manutenção de um projeto. Evitar duplicações de código, abstraindo as partes comuns e encapsulando-as em um único lugar, proporciona maior flexibilidade a aplicação. Ao encapsular comportamentos específicos do sistema em um único lugar, facilita alterações futuras nesses [MCLAUGHLIN *et al.*, 2006].

Outra boa prática importante, trata-se de optar pelo relacionamento entre classes através de interfaces (*i.e.* classes abstratas) ao invés de implementações específicas. A Figura 33 apresenta um exemplo dessa solução aplicada das duas maneiras.

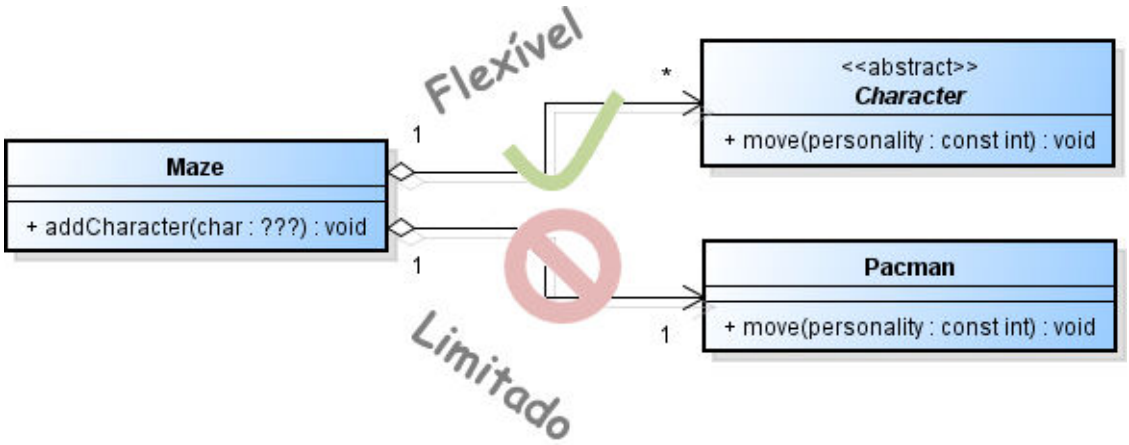


Figura 33 – Interfaces x Implementações, adaptado de [MCLAUGHLIN *et al.*, 2006]

No exemplo ilustrado na Figura 33, a classe *Maze* pode apresentar um acoplamento alto ou baixo, dependendo da abordagem utilizada. Ao desenvolver a classe *Maze* associando-a diretamente com a subclasse *Pacman*, o desenvolvedor estará limitando esse relacionamento, o que inviabiliza a reutilização de tais classes em outras situações.

Outrossim, a associação realizada entre *Maze* e a classe base *Character* se mostra mais flexível, uma vez que a classe *Character* pode ser estendida para outros tipos de personagens, tais como os fantasmas (inimigos do personagem *Pacman*). Essa associação, de fato, mostra-se vantajosa, uma vez que sua concepção pode facilitar possíveis futuras alterações nesse projeto.

Codificar “interfaces”, ao invés de codificar implementações específicas, torna o código mais flexível e, portanto mais fácil de ser estendido. Ao codificar uma classe base chamada de interface, a classe associada a ela trabalhará com todas as subclasses criadas, incluindo as que podem vir a ser criadas no decorrer da vida útil da aplicação [MCLAUGHLIN *et al.*, 2006].

Certamente, tais classes derivadas da “interface” respeitam o modelo de acesso estabelecido por ela (*i.e.* métodos), variando apenas suas implementações internas e encapsuladas. Em termos de OO, uma maneira usual de criar interfaces é por meio de classes com métodos virtuais, os quais podem e até mesmo deveriam ser (tanto quanto possível) métodos virtuais puros (*i.e.* sem implementação definida), visando o chamado polimorfismo.

De modo geral, a utilização de tais boas práticas proporciona maior manutenibilidade e extensibilidade na concepção de *software*, uma vez que os

módulos de *software* seguem regras simples, as quais os tornam mais coesos e menos interdependentes.

2.5.3 Princípios de projeto

A utilização de boas práticas na concepção de *software* auxilia na eliminação dos sintomas de um projeto mal estruturado apresentados anteriormente. Ademais, ao longo dos anos, alguns princípios de projeto foram propostos de maneira a formalizar tais boas práticas, de modo a facilitar a comunicação entre os desenvolvedores.

Um princípio de projeto é uma técnica que pode ser aplicada no projeto ou na escrita do código de maneira a torná-lo mais manutenível, flexível e extensível [MCLAUGHLIN *et al.*, 2006]. Ainda, Martin e Martin (2006) apresentam em seu livro, princípios de projeto que contribuem para eliminação de tais sintomas. Os princípios mais relevantes são conhecidos pelo acrônimo mnemônico SOLID, que são tratados nas subseções subordinadas subsequentes.

2.5.3.1 *Single Responsibility Principle*

O princípio de projeto *Single Responsibility Principle* (*SRP*) exprime a premissa de que cada classe/objeto deve possuir apenas e somente uma razão para ser criada e/ou modificada. Tal razão deve ser totalmente encapsulada na estrutura dessa classe, de forma que todas as suas funcionalidades se alinhem de maneira a atender o objetivo único definido para a classe em questão [MCLAUGHLIN *et al.*, 2006].

Esse princípio foi primeiramente descrito no trabalho de Tom DeMarco (1979) sob o nome de coesão. Em seu trabalho, DeMarco definiu o termo coesão como o relacionamento funcional entre os elementos de um módulo. Martin (2006a), por sua vez, modificou levemente o significado do princípio e relacionou o termo coesão com as forças que causam uma classe a ser modificada [MARTIN, 2006a].

O diagrama de classes ilustrado na Figura 34 apresenta um exemplo de uma classe com duas responsabilidades.

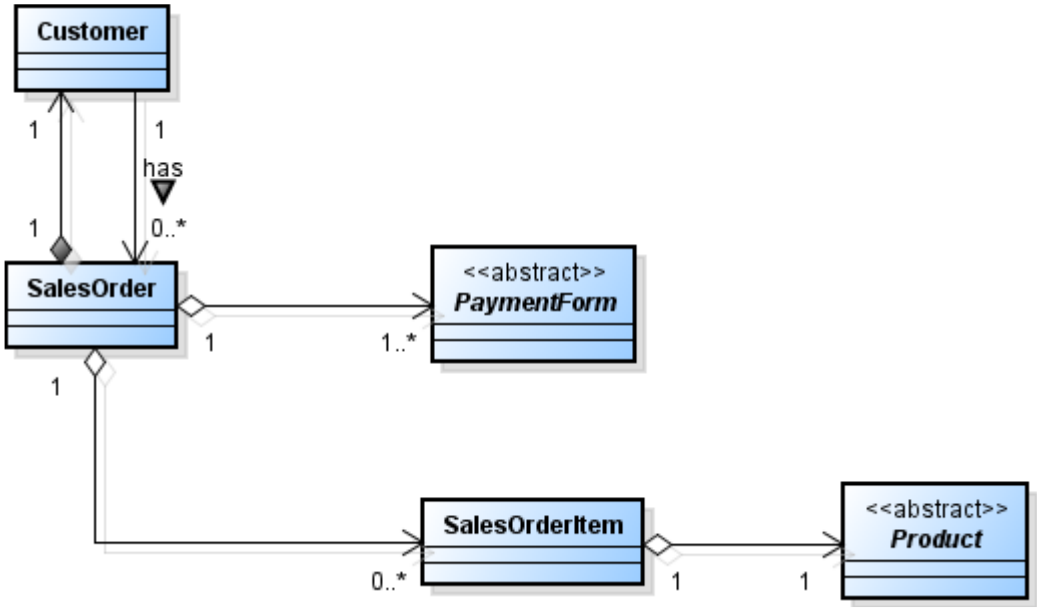


Figura 34 – Exemplo de uma classe com duas responsabilidades

Considerando o diagrama exposto na Figura 34, a classe *SalesOrder* possui duas responsabilidades. A primeira se refere ao armazenamento de dados de um pedido (*i.e.* dados do cliente e forma de pagamento). A segunda, por sua vez, se refere ao gerenciamento de inclusão e remoção de itens de pedido sob uma estrutura de dados.

De maneira a resolver o problema mencionado, as duas responsabilidades presentes na classe *SalesOrder* deveriam ser separadas em classes distintas. Para isso, uma nova classe foi proposta (*SalesOrderItemsList*), com o objetivo de tratar da segunda responsabilidade mencionada. O diagrama exposto na Figura 35 ilustra a separação de responsabilidades aplicada ao problema em questão.

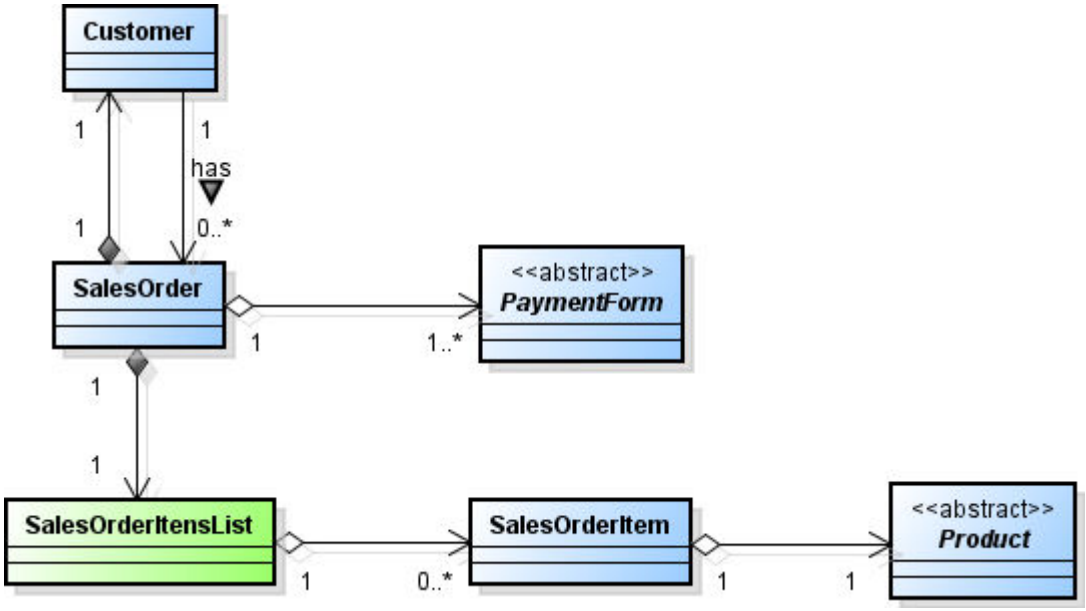


Figura 35 – Exemplo de separação de responsabilidades

Com a criação da classe *SalesOrderItemsList*, a classe *SalesOrder* se limita a única e exclusiva responsabilidade de armazenar os dados de um pedido. Nota-se que apesar de continuar armazenando os itens de um pedido, a responsabilidade de tratar das inclusões e remoções de itens é totalmente delegada à essa nova classe proposta.

A separação de duas responsabilidades em classes distintas proporciona que cada classe seja alterada sem interferir no funcionamento da outra. Desta forma, cada classe terá apenas e somente uma razão para ser alterada. Ademais, uma classe com mais de uma razão para ser modificada apresenta problemas de forte acoplamento. Este tipo de acoplamento tende a gerar problemas de projeto, como o sintoma de Fragilidade, o que leva uma alteração a impactar em diferentes módulos do sistema [MARTIN e MARTIN, 2006].

2.5.3.2 *Open-Closed Principle*

No princípio de projeto *Open-Closed Principle* (*OCP*), entidades de *software* (*e.g.* classes, módulos, funções etc.) deveriam ser abertas para extensão, mas fechadas para modificação [MEYER, 1988]. Em outras palavras, a implementação de módulos deveria possibilitar a extensão de seus comportamentos sem resultar em modificações no código existente.

Quando uma única alteração resulta em uma cascata de alterações entre módulos dependentes, o projeto apresenta o sintoma de Rigidez. O princípio *OCP* aconselha a refatoração do sistema de tal maneira que alterações pontuais não levem a modificações em outras partes do código. Ao ser aplicado corretamente, o princípio determina que modificações adicionais sejam apenas incrementais, com a adição de novas entidades, sem a necessidade de alterar as entidades existentes [MARTIN e MARTIN, 2006].

O mecanismo principal por trás desse princípio é a abstração e o polimorfismo. Criar abstrações que são fixas, porém com um grupo sem restrições de possíveis comportamentos. As abstrações são representadas por classes bases abstratas, enquanto o grupo de comportamentos é representado por todas as possíveis classes derivadas [MARTIN, 2006b].

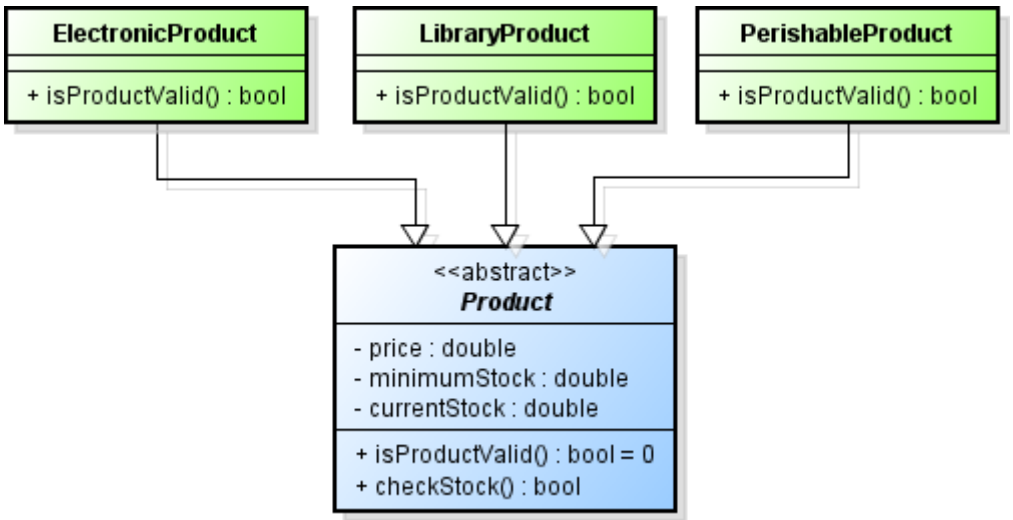


Figura 36 – Exemplo do princípio *OCP* aplicado no sistema de pedido de vendas

Conforme apresenta o diagrama de classes ilustrado na Figura 36, o sistema de pedido de vendas apresenta uma interface abstrata para categorias de produtos, a qual é responsável por comportar as características genéricas dos produtos (*i.e.* preço, estoque mínimo e estoque atual), bem como seus métodos genéricos (*i.e.* validações do produto e estoque). Ademais, é possível observar a existência do método abstrato *isProductValid* na classe *Product*, o qual possui diferentes implementações dependendo do tipo do produto implementado, como por exemplo, verificar a data de validade de produtos perecíveis. Desta forma, novas classes de categoria de produtos com suas particularidades poderiam ser adicionadas ao

sistema sem a necessidade de alterar a estrutura interna das demais classes existentes.

O polimorfismo presente na implementação do método *isProductValid* nesse exemplo, evita que o *software* ao ser estendido necessite de estruturas condicionais (*i.e. if-then* e/ou *switch-case*) para tratar das particularidades de cada uma das categorias de produtos existentes. Com isso, não existe a necessidade de procurar por toda a aplicação, lugares que precisam ser alterados, para que conformem com a nova categoria criada. A solução, desta forma, não apresenta o sintoma de Fragilidade, além de não apresentar o sintoma de Imobilidade, uma vez que as classes podem ser facilmente migradas para outro projeto [MARTIN, 2006b].

Entretanto, a criação exagerada de abstrações pode levar ao sintoma de Complexidade Desnecessária. Uma boa prática, neste sentido, é o de criar abstrações apenas quando ocorrer uma necessidade específica que exija a existência de tal abstração [MARTIN e MARTIN, 2006].

De muitas maneiras, o princípio *OCP* é o coração do projeto orientado a objetos. A conformidade com esse princípio é o que rende maiores benefícios reivindicados pela tecnologia orientada a objetos: flexibilidade, reusabilidade, e manutenibilidade. No entanto, a conformidade com esse princípio não é alcançada simplesmente com a utilização de uma linguagem de programação orientada a objetos. Também não é uma boa ideia aplicar abstrações desenfreadamente em todas as partes da aplicação. Pelo contrário, a utilização desse princípio requer uma dedicação por parte dos desenvolvedores para aplicar abstração apenas às partes do programa que apresentarem mudanças frequentes [MCLAUGHLIN *et al.*, 2006].

2.5.3.3 *Liskov Substitution Principle*

No princípio de projeto *Liskov Substitution Principle* (*LSP*), classes derivadas devem ser capazes de substituir suas classes bases, sem alterar abruptamente o comportamento (*i.e.* tarefa realizada) esperado por essas. Mais formalmente, o princípio *LSP* é uma definição específica de uma relação de subtipagem, chamada de *strong behavioral subtyping* (ou forte subtipagem comportamental, em português), que foi introduzida inicialmente por Barbara Liskov (1988). Tal princípio representa a relação semântica, e não apenas sintática entre duas classes, com a

intenção de garantir a interoperabilidade de tipos em uma hierarquia em particular [MARTIN e MARTIN, 2006].

As funções que usarem ponteiros ou referências para classes base devem ser capazes de usar instâncias de classes derivadas dessa, sem conhecê-las de fato [LISKOV, 1988]. A importância desse princípio se torna evidente quando considerado as consequências geradas pela sua violação. Violar o princípio *LCP* frequentemente resulta no uso de checagem de tipo em tempo de execução, o que consequentemente viola o princípio *OCP*. Frequentemente, uma estrutura condicional explicita é usada para determinar o tipo de um objeto de tal forma que o comportamento apropriado para aquele objeto possa ser selecionado [MARTIN, 2006c]. O Algoritmo 2 apresenta um exemplo de violação dos princípios *LSP* e *OCP*.

Claramente, as linhas de código 6 e 22 expostas no Algoritmo 2 violam o princípio *OCP*, pois tratam de um tipo específico (*Character*) de classe/objeto derivado da classe *Element*. Outrossim, o algoritmo viola o princípio *LSP*, visto que os métodos *getX()* e *getY()* dos elementos dinâmicos (*Character*) não representam a atual posição destes no labirinto, necessitando um método específico (*i.e. checkDynamicColision*) para checar colisões dinâmicas baseado nas intenções de movimentação dos personagens. Os elementos estáticos de um labirinto (*i.e. Corners, Dots e Walls*), por sua vez, são tratados pelo método (*i.e. checkStaticColision*), comparando a posição intencional do personagem em questão com a posição fixa dos elementos estáticos no labirinto.

```
 1 void handleColisions() {
 2   Character* character;
 3   Element* element;
 4   for (int i = 0; i < eleList->getSize(); i++) {
 5     if (eleList->getElement(i)->getType() == Element::CHARACTER) {
 6       character = dynamic_cast<Character*>(eleList->getElement(i));
 7       for (int j = 0; j < eleList->getSize(); j++) {
 8         element = eleList->getElement(j);
 9         if (element->getType() == Element::CHARACTER) {
10           if (character != element) {
11             character->checkDynamicColision(element);
12           }
13         } else {
14           character->checkStaticColision(element);
15         }
16       }
17     }
18   }
19 }
20
21 void Character::checkDynamicColision(Element* element) {
```

```
22    Character* character = dynamic_cast<Character*>(element);
23    if ( (this->getX() + this->getIntentX()) ==
24         (character->getX() + character->getIntentX()) &&
25         (this->getY() + this->getIntentY()) ==
26         (character->getY() + character->getIntentY()) ) {
27      EventManager->getInstance()->newColision(this, character);
28    }
29  }
30
31  void Character::checkStaticColision(Element* element) {
32    if ( (this->getX() + this->getIntentX()) == element->getX() &&
33         (this->getY() + this->getIntentY()) == element->getY() ) {
34      EventManager->getInstance()->newColision(this, element);
35    }
36  }
```

Algoritmo 2 – Exemplo de violação dos princípios *LSP* e *OCP*

A solução ideal para este caso seria criar duas categorias de elementos (*i.e.* subclasses), uma para os que se apresentam estáticos e outra para os que apresentam comportamentos dinâmicos. A Figura 37 apresenta o diagrama de classes que ilustra a solução para o problema, obedecendo ao princípio *LSP*.

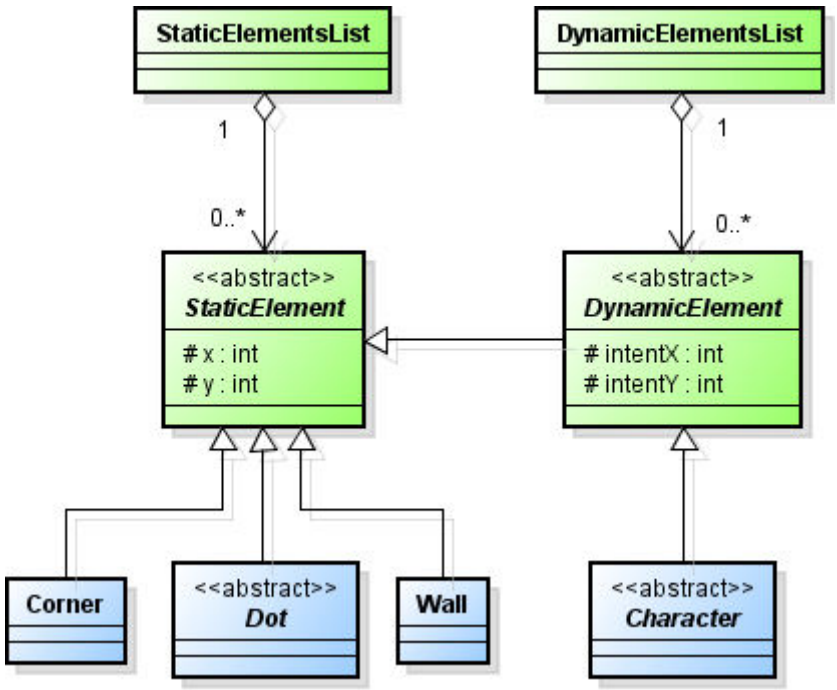


Figura 37 – Exemplo do princípio *LSP* aplicado no simulador do jogo *Pacman*

Conforme ilustra a Figura 37, os elementos dinâmicos devem ser separados dos elementos estáticos, mas não completamente, pois existem características que ambas as categorias apresentam em comum (*x* e *y*). A solução se completa ao possuir duas listas de elementos, uma para elementos estáticos e outra para elementos dinâmicos, evitando conversões de tipos em tempo de execução

(*dynamic_cast* presente nas linhas 6 e 22 do Algoritmo 2), respeitando assim, o princípio *OCP*.

O princípio *LSP* é atendido somente quando tipos derivados são completamente substituíveis por seus tipos base em funções que usam referências desses. Desta forma, respeitando esse princípio, é possível criar hierarquias de classes coesas, que apresentem comportamentos fidedignos aos que foram designados para exercer [MCLAUGHLIN *et al.*, 2006].

2.5.3.4 *Interface Segregation Principle*

No princípio de projeto *Interface Segregation Principle* (*ISP*), clientes não deveriam ser forçados a depender de métodos que não usam. Quando tais clientes são forçados a esse tipo de dependência, esses são sujeitos a mudanças relacionadas com esses métodos. Isso resulta em um acoplamento negligente entre todos os clientes. Dito de outra forma, quando um cliente depende de uma classe que contém métodos que não são usados, mas que outros clientes usam, aquele cliente específico será afetado por mudanças que outros clientes forçam nessa classe [MARTIN, 2006d].

Esse princípio lida com as desvantagens de interfaces “gordas”. Classes que possuem interfaces não coesas são consideradas “gordas”. Não obstante, as interfaces de uma classe podem ser quebradas em grupos de métodos. Cada grupo serve a um diferente conjunto de clientes. Desta forma, alguns clientes usam um grupo de métodos, e outros clientes usam outros grupos [MARTIN e MARTIN, 2006].

O princípio *ISP* reconhece que há objetos que requerem interfaces não coesas, no entanto, sugere que os clientes não devem conhecer sobre elas como uma única classe. Em vez disso, clientes que precisarem de funcionalidades extras, devem depender de mais interfaces (coesas), ao invés de poluir todos os demais clientes com dada necessidade [MARTIN e MARTIN, 2006]. A Figura 38 mostra um exemplo de dependência que infringe o princípio *ISP*.

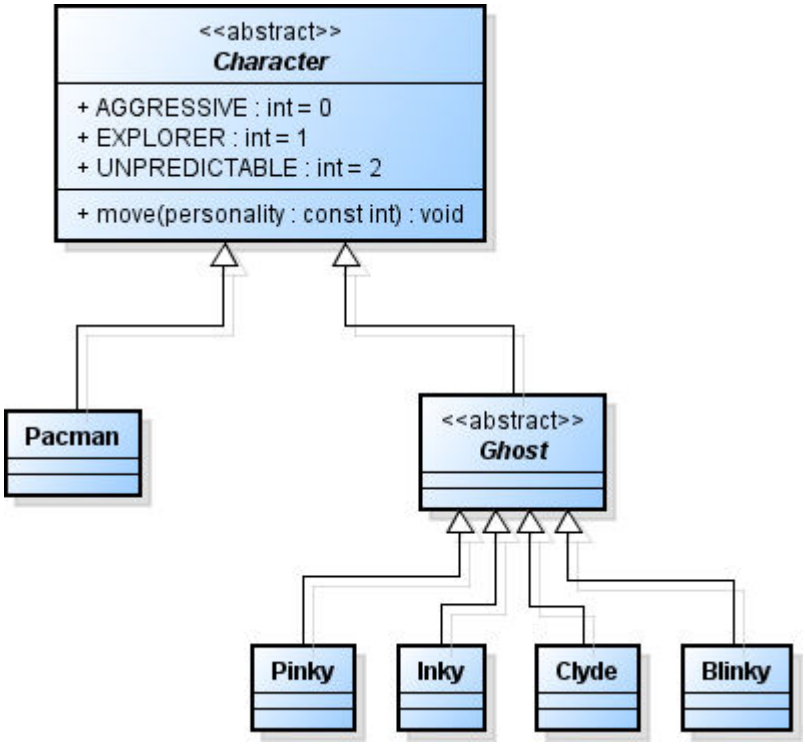


Figura 38 – Exemplo de violação do princípio *ISP*

Conforme ilustra o exemplo exposto no diagrama de classes da Figura 38, o problema está na dependência da classe *Character* com o parâmetro *personality* e o impacto deste na execução do método *move*. Nem todas as variações de *Character* precisam de personalidades definidas, como no caso do personagem *Pacman*. A personalidade deste, isto é, suas ações no labirinto são predefinidas pelo jogador, conforme explicitado na Subseção 2.4.1.

Certamente, a abstração original de *Character* não deveria ter relação direta com *personality*, uma vez que nem todas as classes derivadas de *Character* necessitam de uma personalidade definida, o que seguramente infringe o princípio *LSP*. Ademais, novas classes que herdarem de *Character* terão de tratar a definição de personalidade, mesmo que esta não seja usada. Isto leva aos sintomas de Complexidade Desnecessária e Repetições Inúteis.

Esse é um exemplo de poluição de interface, uma síndrome comum em linguagens estaticamente tipadas como C++, C# e Java. A interface de *Character* foi poluída com uma funcionalidade que não é requerida. Ela foi forçada a incorporar esta funcionalidade somente para o benefício de uma de suas subclasses (*i.e. Ghost*). Se esta prática se tornar comum em um determinado projeto, cada vez que uma classe derivada necessitar de uma nova funcionalidade, esta será adicionada à

classe base. Isto poluirá a interface dessa classe base, tornando-a “gorda” [MARTIN, 2006d].

Ainda, a implementação baseada nos estados de parâmetros necessita de estruturas de decisão para validá-los e trata-los corretamente, o que contribui para a formação do sintoma de rigidez. A solução para este problema se dá através da separação da funcionalidade em uma classe/interface independente, fornecida apenas aos clientes interessados nesta. As classes não precisam necessariamente herdar características e implementar todas as funcionalidades em sua própria estrutura. Ao invés disso, estas podem delegar funcionalidades para outra classe associada, agregada ou composta a elas. Com isso, é possível que classes utilizem o comportamento de uma família de outras classes, e alterem este comportamento em tempo de execução [MCLAUGHLIN *et al.*, 2006]. A Figura 39 ilustra um exemplo de separação de funcionalidades por delegação.

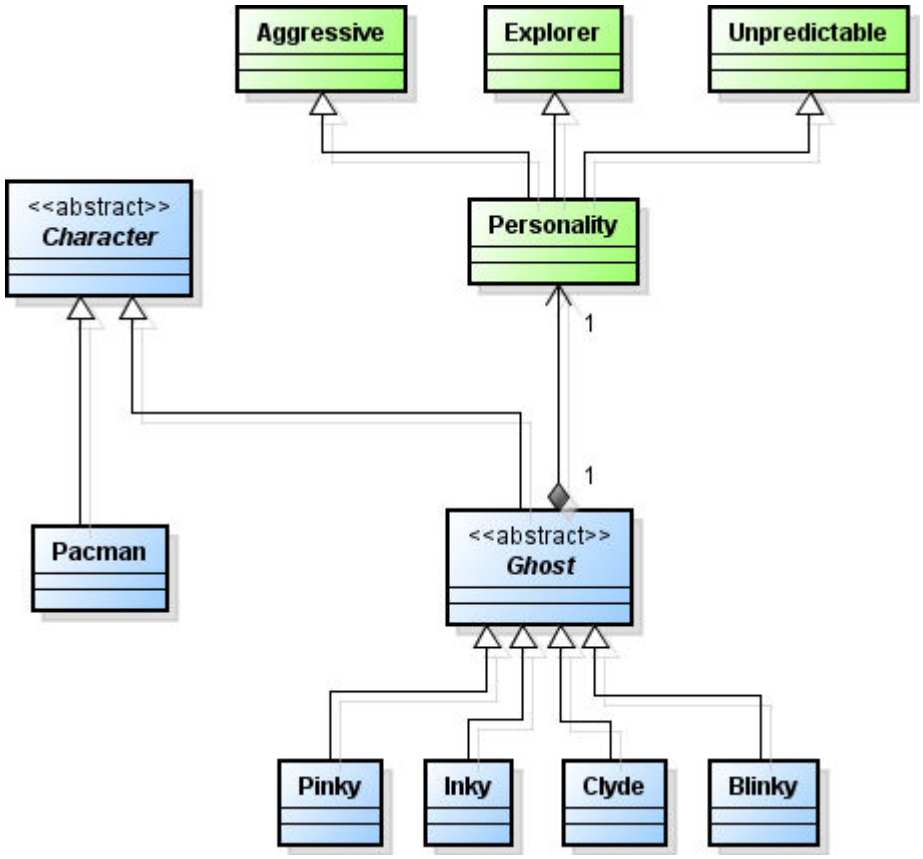


Figura 39 – Exemplo de separação de funcionalidades por delegação

A solução apresentada na Figura 39 obedece ao princípio *ISP* e previne o acoplamento de subclasses de *Character* com a funcionalidade que define suas

personalidades. Nesta solução, mesmo que uma alteração em *Personality* fosse feita, nenhuma das subclasses de *Character* seriam afetadas. Ademais, a classe *Ghost* não apresenta a mesma interface que a classe *Pacman*.

Classes “gordas” causam acoplamentos prejudiciais e indesejáveis entre seus clientes. Quando um cliente força a alteração de uma classe “gorda”, todos os outros clientes são afetados. Desta forma, clientes devem ser dependentes apenas de funcionalidades que eles utilizam.

Esse cenário ideal pode ser alcançado quebrando a interface de uma classe “gorda” em muitas interfaces específicas. Cada interface específica declara somente as funcionalidades para seu cliente particular ou grupo de clientes invocarem. As classes “gordas” podem então herdar todos as interfaces específicas e implementá-las. Isso quebra a dependência dos clientes nos métodos que não invocam e permite com que os clientes sejam independentes uns dos outros [MARTIN e MARTIN, 2006].

2.5.3.5 *Dependency Inversion Principle*

No princípio de projeto *Dependency Inversion Principle* (*DIP*), módulos de *software* que encapsulam políticas de alto-nível (*e.g.* objetos centralizadores) não devem depender de módulos de *software* de baixo-nível (*e.g.* objetos colaboradores). Ambos devem depender de abstrações [MARTIN, 2006e].

A razão da existência desse princípio se deve pelo fato de que métodos tradicionais de desenvolvimento de *software*, tais como análise e projeto estruturados, tendem a criar estruturas de *software* nas quais módulos de alto-nível dependem de módulos de baixo-nível. Um dos objetivos desses métodos é definir uma arquitetura que descreva os módulos de alto-nível chamando funções/métodos nos módulos de baixo-nível [MARTIN e MARTIN, 2006].

De acordo com Freeman *et al.* (2004), o princípio *DIP* se parece com a boa prática citada na Subseção 2.4.2 – Não depender de implementações específicas, mas sim de abstrações – ou seja, programar voltado a interfaces. Na verdade, esses são similares; entretanto, o princípio *DIP* faz uma afirmação ainda mais forte em relação a abstrações. O princípio sugere que um componente de alto nível, como por exemplo, a classe *Maze* do simulador do jogo *Pacman*, responsável por

comportar todos os elementos que compõem um labirinto, não deveria depender de componentes de baixo-nível como *Walls*, *Dots* e *Corners*. Ao invés disso, ambos os tipos componentes deveriam depender de abstrações. Tal exemplo é ilustrado na Figura 40.

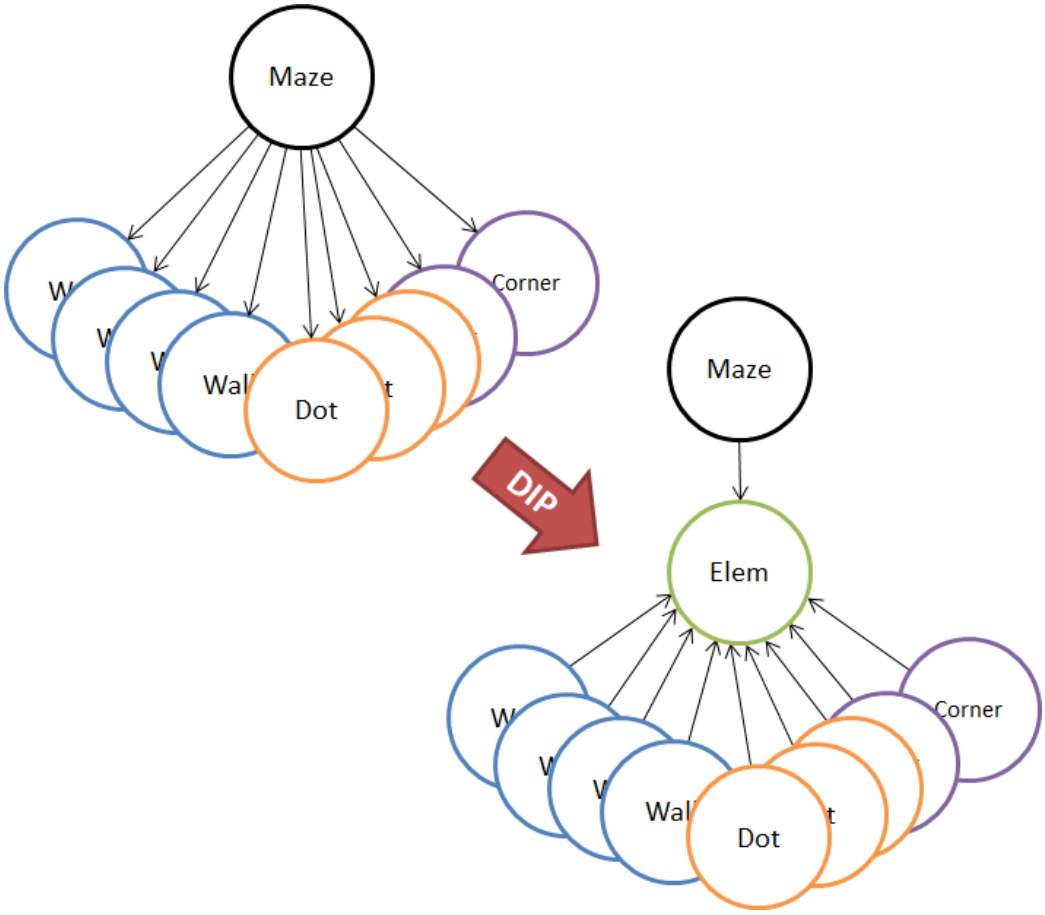


Figura 40 – Inversão de dependências aplicando o princípio *DIP*

Conforme ilustra a Figura 40, no modelo original, o labirinto se mostrava fortemente acoplado aos elementos que constituem sua estrutura. Nesse modelo, o labirinto precisava conhecer diretamente cada uma das classes que compõem sua composição. Ao criar um ponto intermediário entre os elementos de alto-nível e de baixo-nível, as dependências se invertem, o que torna a solução mais flexível. No caso da necessidade de criação de outros elementos distintos, esses necessitariam apenas estender a classe *Element*, usufruindo dos benefícios do polimorfismo.

Todavia, esse princípio por si só não se apresenta como uma solução definitiva para a separação de responsabilidades da classe *Maze*. Tal classe apresenta duas responsabilidades distintas (*i.e.* criar e comportar os elementos que

a compõem), o que, por sua vez, infringe o princípio de projeto *SRP*. Na verdade, o princípio de projeto *DIP* forma a base para um padrão de projeto usualmente aplicado na criação de objetos (*i.e. Abstract Factory)*, o qual quando aplicado corretamente conforma com o princípio *SRP*. Tal padrão é apresentado na Subseção 2.5.2.

Assim como o princípio *DIP* forma a base para o padrão de projeto *Abstract Factory*, os demais princípios de projeto desempenham um papel importante para a composição de grande parte dos padrões de *software* existentes. Neste sentido, é possível mesmo sem conhecer todos os padrões de projeto, desenvolver *software* com qualidade, seguindo as boas práticas mencionadas neste trabalho.

Outrossim, a seção subsequente apresenta os padrões de projeto mais relevantes para a estruturação deste trabalho.

2.6 PADRÕES DE PROJETO

De maneira geral, este trabalho enfatiza a aplicação de princípios de projeto em programas orientados a objetos (OO). Contudo, tais princípios poderiam supostamente ser aplicados a qualquer paradigma de programação, uma vez que apresentam como objetivo básico, a diminuição de acoplamento e o aumento de coesão.

A concepção de sistemas flexíveis, reutilizáveis e manuteníveis em OO não se resume na simples utilização dos conceitos fundamentais da orientação a objetos (*i.e.* abstração, encapsulamento, herança, polimorfismo, coesão e desacoplamento) na estrutura de tais sistemas. A construção de sistemas com tais propriedades não é uma tarefa trivial e muitas vezes pouco intuitiva, que exige normalmente diversos refinamentos ao longo do processo de desenvolvimento. Essas maneiras, muitas vezes, não obvias, de construir sistemas orientados a objetos foram reunidas em um conjunto de padrões denominado de Padrões de Projeto [FREEMAN *et al.*, 2004].

Neste âmbito, padrões de projeto representam um conjunto de soluções genéricas para problemas recorrentes na concepção de programas baseados no POO. Contudo, tais padrões não se apresentam como soluções definitivas para um problema específico, necessitando que suas estruturas sejam adaptadas ao problema em questão.

De acordo com Freeman *et al.* (2004), nem sempre será possível encontrar padrões específicos para um caso em particular. Todavia, cabe ao desenvolvedor refinar seu projeto de acordo com tais padrões.

A utilização de padrões de projeto na concepção de sistemas auxilia os desenvolvedores na estruturação de suas aplicações de maneira que essas se tornem mais flexíveis, a ponto de facilitar a alteração e/ou adição de funcionalidades no processo de desenvolvimento.

Ademais, a utilização desses apresenta como principais vantagens: (a) refletir a experiência de desenvolvedores que tiveram êxito com a aplicação de padrões em seus projetos; (b) prover uma solução pré-arquitetada aplicável a diferentes problemas; (c) prover um vocabulário comum entre os desenvolvedores, acelerando o processo de criação e/ou expansão de soluções computacionais.

Devido à alta granularidade e níveis de abstração, bem como a existência de um número considerável de padrões de projeto, Gamma *et al.* (1995) classificaram e organizaram os principais padrões seguindo dois critérios principais. A Tabela 1, apresenta a classificação dos padrões de projeto, segundo os critérios estabelecidos [GAMMA *et al.*, 1995].

Tabela 1 – Classificação dos padrões de projeto [GAMMA *et al.*, 1995]

		Propósito		
		Criacionais	**Estruturais**	**Comportamentais**
Escopo	**Classe**	*Factory Method*	*Adapter*	*Interpreter* *Template Method*
	Objeto	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Adapter* *Bridge* *Composite* *Decorator* *Facade* *Flyweight* *Proxy*	*Chain of Responsability* *Command* *Iterator* *Mediator* *Memento* *Observer* *State* *Strategy*

O primeiro critério trata-se do propósito para a existência de determinado padrão, refletindo a sua contribuição na solução de um determinado problema. De acordo com esse critério, tais padrões são classificados em três diferentes grupos, que são os padrões criacionais, padrões estruturais e os padrões comportamentais. Os padrões criacionais afetam o processo de criação de objetos; por sua vez, os

padrões estruturais lidam com a composição de classes e objetos; por fim, os padrões comportamentais caracterizam os meios pelos quais classes e objetos interagem e distribuem responsabilidades entre si.

O segundo critério, por sua vez, trata-se do escopo de utilização de um determinado padrão, especificando a aplicação desse, seja na composição de classes ou na instanciação de objetos.

Para os padrões que se aplicam na composição de classes, esses geralmente atuam no tratamento de relacionamentos entre classes e suas subclasses, estabelecendo tais relacionamentos através de herança, o que define seus comportamentos de maneira estática, fixada em tempo de compilação.

Os padrões que se aplicam na instanciação de objetos, por sua vez, lidam com o relacionamento entre objetos, os quais podem ser alterados em tempo de execução e apresentam comportamento dinâmico.

Ainda, padrões criacionais, em nível de classes, delegam parte da criação de objetos para suas respectivas subclasses, enquanto em nível de objetos, delegam tal criação para outro objeto [GAMMA *et al.*, 1995].

Por sua vez, padrões estruturais, em nível de classes, usam herança para a composição de classes, enquanto em nível de objetos, descrevem maneiras de montar objetos [GAMMA *et al.*, 1995].

Por fim, padrões comportamentais, em nível de classes, usam herança para descreverem algoritmos e o fluxo de controle, enquanto em nível de objetos, descrevem como um grupo de objetos coopera para realizar uma tarefa que um objeto único não poderia realizar sozinho [GAMMA *et al.*, 1995].

Neste trabalho, não serão abordados todos os padrões de projeto existentes, mas sim, apenas os mais relevantes para o entendimento do *Framework* PON, bem como seus possíveis desdobramentos em trabalhos futuros. Os padrões abordados neste trabalho são: *Strategy*, *Abstract Factory*, *Observer*, *Singleton*, *Iterator*, *Command*, *Composite* e *Model View Controller* (*MVC*). Esses padrões são apresentados nas subseções seguintes.

2.6.1 *Strategy*

O padrão comportamental *Strategy* define um conjunto de algoritmos relacionados, encapsula cada um deles em classes e os torna intercambiáveis. Ademais, esse padrão permite que o comportamento de um objeto varie em tempo de execução. Cada algoritmo encapsulado e classificado dessa maneira é denominado de estratégia [GAMMA *et al.*, 1995]. A Figura 41 apresenta a estrutura do padrão de projeto *Strategy*.

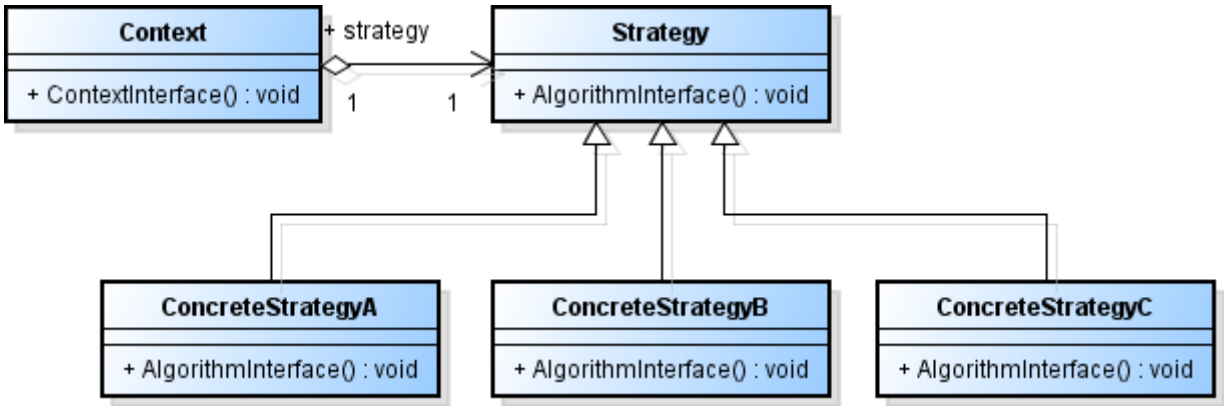


Figura 41 – Estrutura do padrão de projeto *Strategy*, adaptado de [GAMMA *et al.*, 1995]

Conforme ilustra a Figura 41, os participantes desse padrão são [GAMMA *et al.*, 1995]:

- ***Strategy***: define uma interface comum para todos os algoritmos suportados;
- ***ConcreteStrategy***: implementa o algoritmo usando a interface *Strategy*;
- ***Context***: é configurado com um objeto *ConcreteStrategy* e mantém uma referência para um objeto *Strategy*.

A utilização desse padrão prove vantagens como: (a) quebrar um conjunto de algoritmos separados por estruturas condicionais em classes relacionadas, provendo maior coesão e menor acoplamento entre esses; (b) permitir a execução de diferentes comportamentos presentes em classes relacionadas, em diferentes momentos durante a execução da aplicação; (c) possibilitar a criação de novos comportamentos (*i.e.* estratégias) sem interferir na codificação das estratégias já existentes.

Em contra partida, a utilização desde padrão ocasiona certa sobrecarga de comunicação entre as classes *Context* e *Strategy*. Ademais, a interface *Strategy* é compartilhada por todas as classes concretas que a estendem, independente de suas implementações serem triviais ou complexas. Outrossim, a utilização deste padrão tende a aumentar o número de classes/objetos de uma aplicação, o que tende a aumentar a complexidade do sistema [GAMMA *et al.*, 1995].

De modo a exemplificar o uso do padrão *Strategy*, a Figura 42 o apresenta aplicado no simulador jogo *Pacman* sobre o comportamento dos personagens que o compõem.

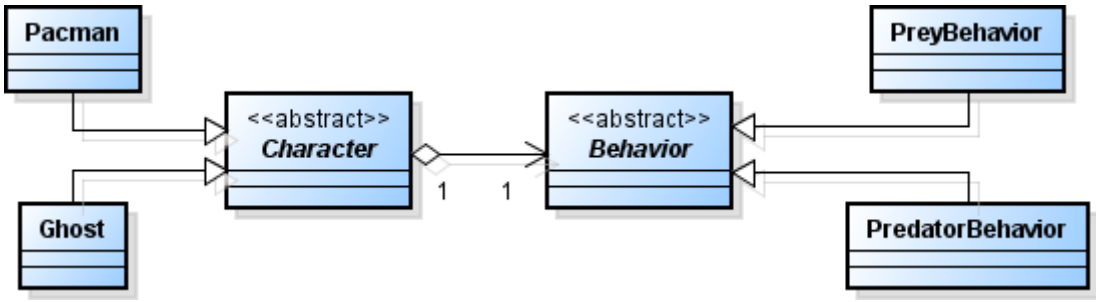


Figura 42 – Exemplo do padrão *Strategy*

Conforme ilustra a Figura 42, o comportamento dos personagens (*i.e. Pacman* ou *Ghosts*) pode ser alterado em tempo de execução por meio da simples troca de comportamentos. No simulador do jogo (Subseção 2.4.1), tal mudança ocorre quando o *Pacman* absorve uma pastilha energizadora, o que inverte os papéis do jogo por um determinado tempo, onde o *Pacman* deixa de apresentar o comportamento de presa e passa a se orientar pelo comportamento de predador.

No exemplo ilustrado, o princípio *SRP* é expresso na separação de funcionalidades e delegação de comportamentos para objetos específicos (*i.e.* comportamentos específicos), ao invés de usar métodos definidos na estrutura da classe *Character* (ou suas subclasses) separados por estruturas condicionais. Ademais, o padrão define uma estrutura ideal para a adição de novos comportamentos, sem a necessidade de alterar a codificação existente, respeitando o princípio *OCP*. Ainda, a estrutura desse padrão respeita todos os demais princípios descritos no trabalho, conforme dito anteriormente.

2.6.2 *Abstract Factory*

O padrão de criação *Abstract Factory* permite com que o cliente use uma interface abstrata para criar um conjunto de produtos (*i.e.* objetos) relacionados sem se preocupar com os produtos concretos que são produzidos de fato. Desta forma, o cliente é desacoplado de qualquer um dos produtos concretos existentes. A utilização desse padrão possibilita com que novos tipos de produtos sejam adicionados sem a necessidade de alterar o código do cliente [FREEMAN *et al.*, 2004]. A Figura 43 apresenta a estrutura do padrão de projeto *Abstract Factory*.

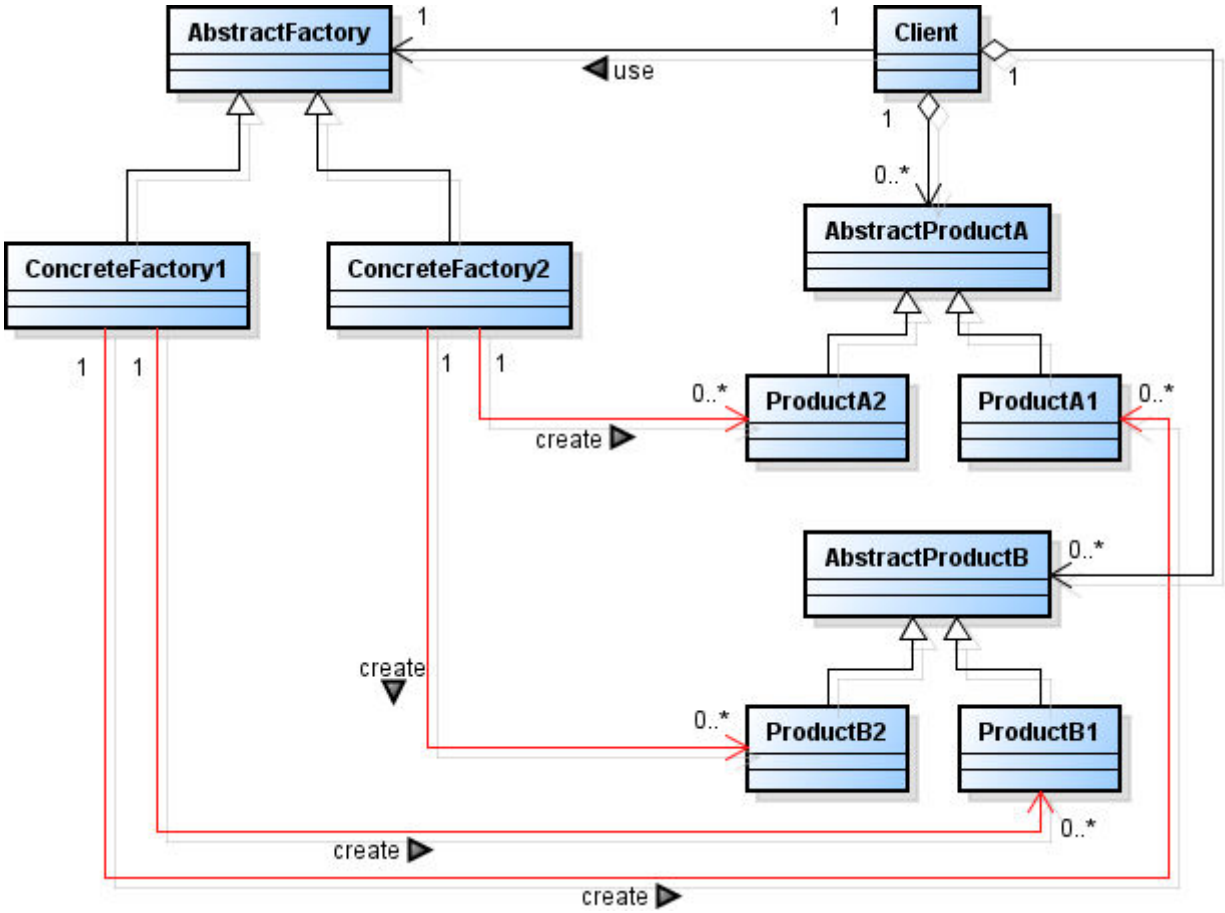


Figura 43 – Estrutura do padrão de projeto *Abstract Factory* [FREEMAN *et al.*, 2004]

Conforme apresenta a Figura 43, os participantes desse padrão são [GAMMA *et al.*, 1995]:

- ***AbstractFactory***: declara uma interface para criação de produtos;

- ***ConcreteFactory***: implementa as operações para criação de produtos concretos (*i.e.* objetos);
- ***AbstractProduct***: declara uma interface para um tipo de produto;
- ***ConcreteProduct***: define um produto para ser criado pela fábrica concreta correspondente;
- ***Client***: usa a interface da fábrica abstrata para a criação de objetos concretos compatíveis com as interfaces (*i.e.* produtos abstratos) conhecidas pelo cliente.

A utilização desse padrão prove vantagens como: (a) desacoplar implementações entre classes concretas interdependentes; (b) permitir a instanciação de diferentes famílias de objetos de acordo com a fábrica escolhida; (c) possibilitar a criação de novas famílias de objetos sem interferir na codificação existente.

Outrossim, conforme mencionado na Subseção 2.4.3.5, esse padrão tem sua base formada pela inversão de dependências, isto é, pela aplicação do princípio *DIP*. A inversão de dependências ocorre no interfaceamento de produtos agregados ao cliente, onde esse último deixa de depender de produtos concretos e passa a depender apenas de interfaces para produtos abstratos. A solução se completa, ao delegar a função de criação de objetos para uma classe específica, ou seja, a fábrica de produtos, tornando a interface do cliente mais limpa e coesa.

Em contra partida, a criação de novas famílias de produtos a partir da extensão de uma fábrica abstrata nem sempre é uma tarefa simples. Em alguns casos, isso pode ocasionar inclusive mudanças na estrutura da classe abstrata, o que implicaria em mudanças em todas as classes concretas existentes [GAMMA *et al.*, 1995].

De modo a exemplificar o uso do padrão *Abstract Factory*, a Figura 44 o apresenta aplicado no simulador do jogo *Pacman sobre* a criação de elementos que compõe o labirinto do mesmo.

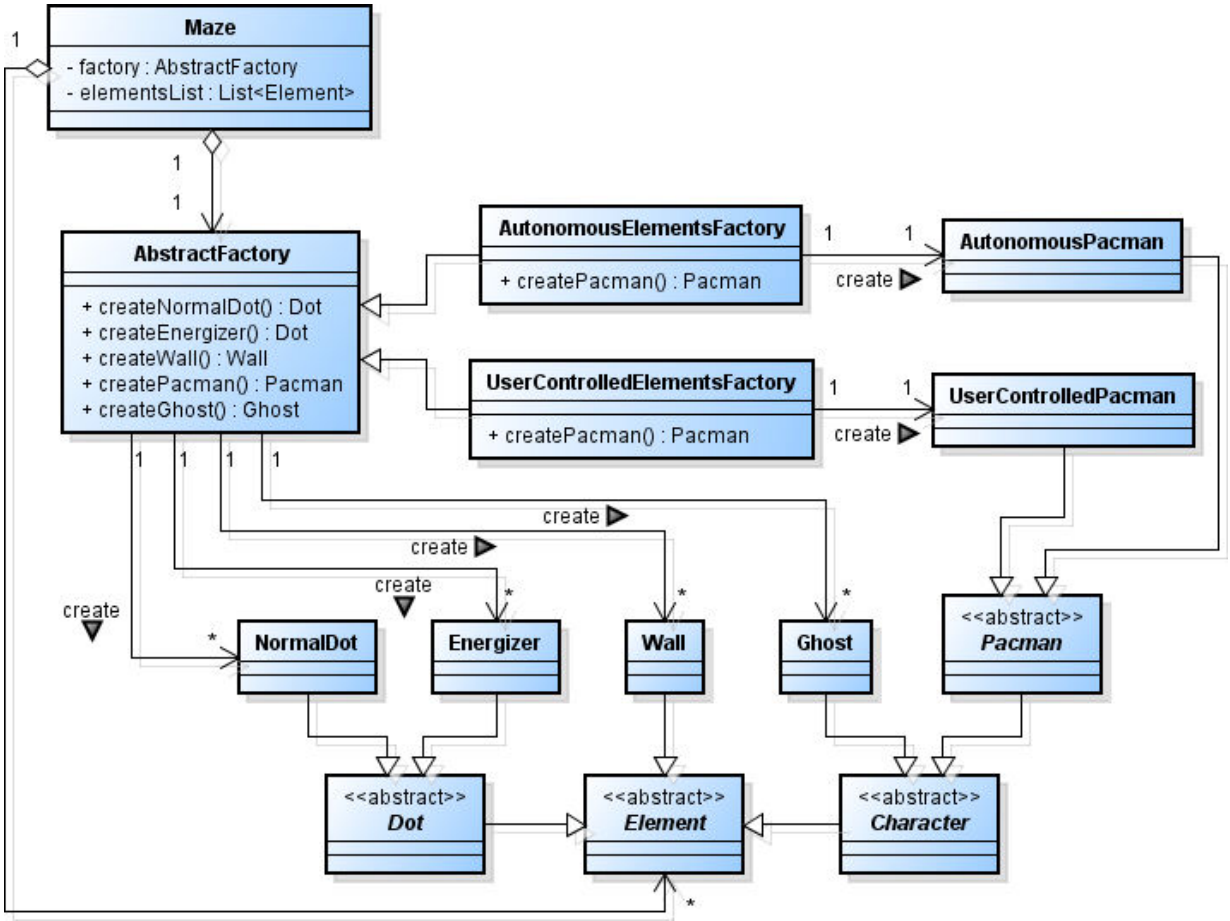


Figura 44 – Exemplo do padrão *Abstract Factory*

Conforme ilustra a Figura 44, a instanciação de objetos para compor o labirinto se dá através de métodos abstratos definidos em uma interface denominada *AbstractFactory*. Tal fábrica define uma família de objetos relacionados que serão instanciados no momento da construção do labirinto.

As fábricas concretas, por sua vez, instanciam os produtos concretos que pertencem a diferentes famílias de produtos. Para criar um elemento, o cliente usa uma dessas fábricas. Desta forma, ele não precisa conhecer nem instanciar os objetos diretamente.

A classe *Maze* representa o cliente em questão e é composta por uma fábrica abstrata e por uma lista de elementos (*i.e.* produtos abstratos). Essa classe, em tempo de execução, através de uma instância de uma fábrica concreta, solicita a criação de elementos concretos.

O padrão *Abstract Factory* possibilita a criação de diferentes tipos de objetos, baseados na fábrica concreta delegada para essa função. No exemplo, a fábrica de elementos padrão tem a responsabilidade de criar os elementos comuns

aos labirintos, tanto para versão do *Pacman* autônomo, quanto para a versão do *Pacman* controlado por um usuário.

2.6.3 *Observer*

O padrão comportamental *Observer* define uma relação de um-para-muitos entre objetos de tal forma que quando um objeto muda de estado, todos os objetos interessados em tal mudança são notificados e atualizados automaticamente [GAMMA *et al.*, 1995]. Ademais, esse padrão ajuda a sincronizar o estado de componentes colaboradores. Para isso, realiza uma propagação unilateral de notificações, na qual o observado notifica um ou mais observadores em quaisquer mudanças ocorridas em seu estado interno. Ainda, esse padrão representa um ponto importante na programação orientada a eventos [SCHMIDT *et al.*, 2000]. A Figura 45 apresenta a estrutura do padrão de projeto *Observer*.

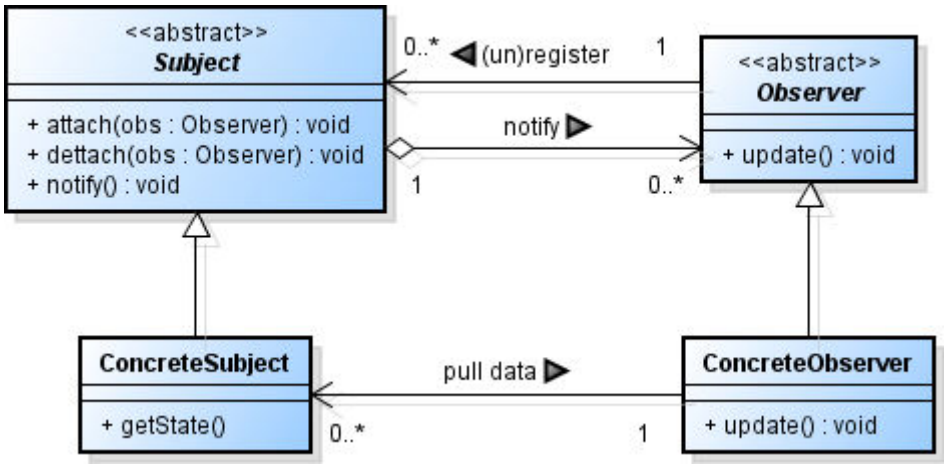


Figura 45 – Estrutura do padrão de projeto *Observer*, adaptado de [GAMMA *et al.*, 1995]

Conforme apresenta a Figura 45, os participantes desse padrão são [GAMMA *et al.*, 1995]:

- ***Subject***: prove uma interface para anexar ou desanexar *Observers*, e notificar todos os *Observers* anexos a cada nova mudança em seu estado;
- ***Observer***: define uma interface de atualização para os objetos que devem ser notificados pelas mudanças ocorridas no *Subject*;

- ***ConcreteSubject***: armazena o estado de interesse para os *ConcreteObservers*;
- ***ConcreteObserver***: mantem uma referencia para o *ConcreteSubject*, e implementa a interface de atualização para manter seu estado consiste com o do *ConcreteSubject*.

A utilização desse padrão prove vantagens como: (a) baixo acoplamento entre os objetos observadores e os observados; (b) permite gerenciar a relação entre os objetos observadores e os observados em tempo de execução; (c) proporciona autonomia aos objetos observadores sobre considerar ou ignorar determinada atualização ocorrida no objeto observado.

Em contra partida, esse padrão pode provocar certa sobrecarga de processamento na execução do sistema como um todo, devido a atualizações inesperadas nos objetos observadores. Tal problema é agravado pelo fato de atualizações simples não proverem detalhes sobre as mudanças ocorridas no objeto observado, as quais poderiam ter pouco ou nenhum impacto no estado dos objetos observadores. Com a falta desse controle, tais objetos seriam obrigados a se atualizarem sobre quaisquer mudanças no objeto observado [GAMMA *et al.*, 1995].

De modo a exemplificar o uso do padrão *Observer*, a Figura 46 o apresenta aplicado no simulador do jogo *Pacman* sobre o mecanismo de gerenciamento e tratamento de eventos.

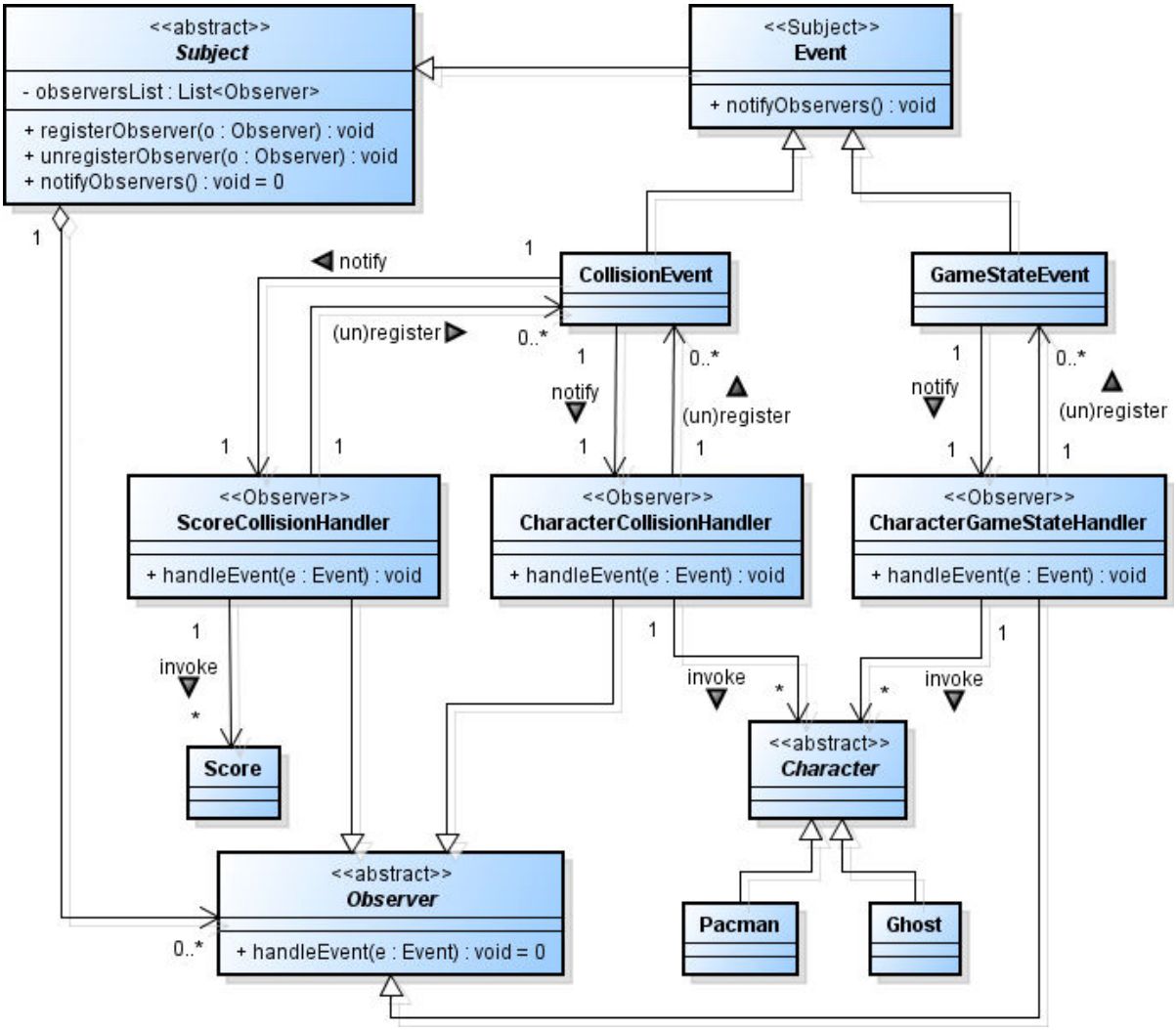


Figura 46 – Exemplo do padrão *Observer*

No exemplo ilustrado na Figura 46, tanto a classe *Subject*, quanto a classe *Observer*, são classes abstratas que representam funcionalidades específicas e bem-definidas, respeitando os princípios de projeto *SRP* e *ISP*. Ainda, os relacionamentos entre tais classes ocorrem em um nível abstrato, o que representa o cumprimento do padrão *DIP*.

Ademais, o princípio *SRP* é materializado na criação de *Events* e *Handlers* específicos. Os *Events* possuem a responsabilidade de armazenar os dados referentes ao evento em questão, delegando a responsabilidade de notificar todos os interessados sobre o ocorrido à sua classe base. Os *Handlers*, por sua vez, tratam de tais eventos, invocando métodos pertinentes nos objetos apropriados.

Outrossim, o princípio *OCP* é expresso na estrutura dos *Handlers*, os quais possuem uma estrutura bem-definida e tratam de eventos específicos, sem a necessidade de alterar o código existente. Além disso, o princípio *LSP* é valido, uma

vez que para a funcionalidade específica de notificações, a classe *Event* pode ser substituída pela classe *Subject* sem interferir no funcionamento da aplicação. Neste sentido, outra implementação válida para o relacionamento e delegação de funções entre *Events* e sua classe base *Subject*, seria a delegação por meio de agregação, o que traria mais transparência para o princípio *SRP*.

O padrão *Observer* é um dos padrões de projeto mais utilizados devido à possibilidade de redução de acoplamento entre objetos de uma aplicação. Esse padrão é massivamente empregado em grandes projetos, como na estrutura interna do JDK (*Java Development Kit*) [FREEMAN *et al.*, 2004].

2.6.4 *Singleton*

Usualmente, relações entre classes e instâncias seguem um padrão de relacionamento um-para-muitos. Ademais, é possível criar muitas instâncias de muitas classes. Tais instâncias são criadas quando necessário e dispensadas quando sua utilidade acaba. Elas são construídas e destruídas sob um fluxo de alocações e desalocações de memória constante. Entretanto, algumas classes deveriam possuir apenas uma única instância.

Tal instância deve surgir quando o sistema iniciar e deve desaparecer somente quando o sistema terminar. Esses objetos são algumas vezes as raízes de uma aplicação. Se mais de uma raiz é criada, o fluxo de execuções pode tomar um rumo totalmente contrário ao desejado. Neste sentido, o padrão de criação *Singleton* assegura que uma classe possuirá apenas uma única instância, provendo acesso a essa de forma global e uniforme [MARTIN e MARTIN, 2006]. A Figura 47 apresenta a estrutura do padrão de projeto *Singleton*.

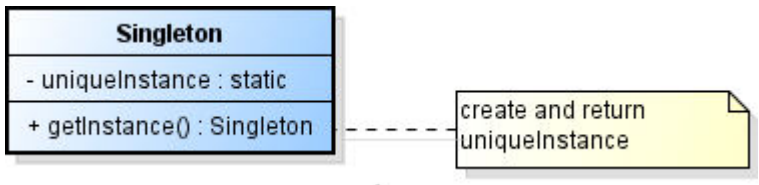


Figura 47 – Estrutura do padrão de projeto *Singleton* [GAMMA *et al.*, 1995]

Conforme apresenta a Figura 47, o único participante desse padrão é a classe *Singleton*, responsável por criar sua própria e única instância e retorná-la para os objetos que invocarem o método *getInstance* [GAMMA *et al.*, 1995].

A utilização desse padrão prove vantagens como: (a) acesso controlado à instância única; (b) controle sobre como e quando os clientes acessam tal instância; (c) maior flexibilidade em comparação a variáveis globais e métodos estáticos.

Ademais, o padrão *Singleton* assegura que um e somente um objeto seja instanciado de uma dada classe. O padrão *Singleton* também prove acesso global a esse objeto, assim como uma variável global, porém sem as desvantagens dessa. Diferentemente de variáveis globais, o padrão só cria uma instância da classe quando requisitado, sem a necessidade de definí-la no início do programa. Além disso, variáveis globais podem ter seu estado substituído em qualquer parte do código no decorrer da execução do programa, o que não ocorre nesse padrão. O padrão tem total controle sobre a criação e substituições de instâncias em seu mecanismo interno [FREEMAN *et al.*, 2004].

De modo a exemplificar o uso do padrão *Singleton*, a Figura 48 o apresenta aplicado no simulador do jogo *Pacman* sobre o mecanismo de gerenciamento de eventos.

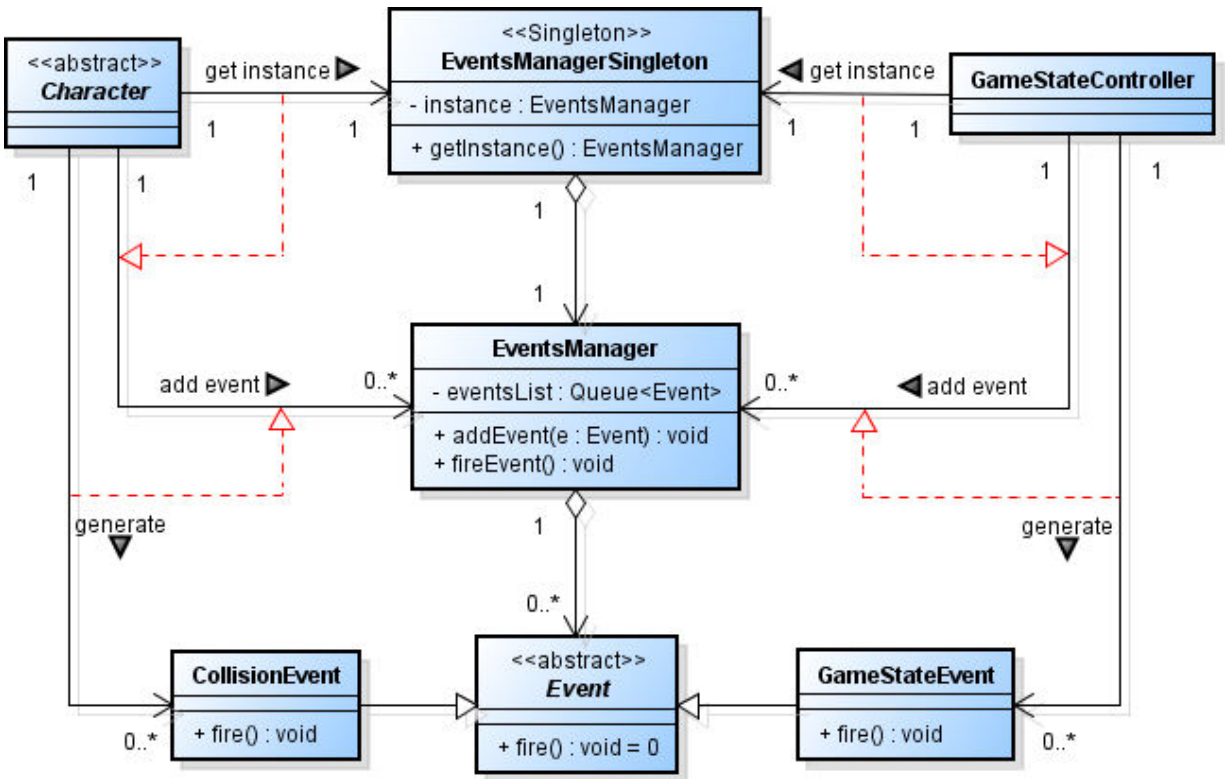


Figura 48 – Exemplo do padrão *Singleton*

A maneira com que o padrão *Singleton* foi projetado no exemplo ilustrado na Figura 48, mostra que as respectivas classes respeitam o princípio *SRP*. Isso ocorre na separação de responsabilidades em duas classes, onde a classe *EventsManagerSingleton* controla a criação e o acesso de uma instância única da classe *EventsManager* que, por sua vez, gerencia a execução de eventos no jogo *Pacman*.

Ainda, no exemplo ilustrado, é possível observar que muitas das funcionalidades, tais como gerar eventos e adicioná-los à classe *EventsManager*, são realizadas inicialmente pela chamada do método *getInstance*, que fornece acesso global a classe responsável por tais funcionalidades.

2.6.5 *Iterator*

O padrão comportamental *Iterator* prove um meio de acessar um conjunto de elementos armazenados em um objeto específico (*e.g.* lista de elementos) sequencialmente sem expor suas representações internas. Ademais, esse padrão geralmente fornece meios diversos para o percorrimento de tal estrutura [GAMMA *et al.*, 1995]. A Figura 49 apresenta a estrutura do padrão de projeto *Iterator*.

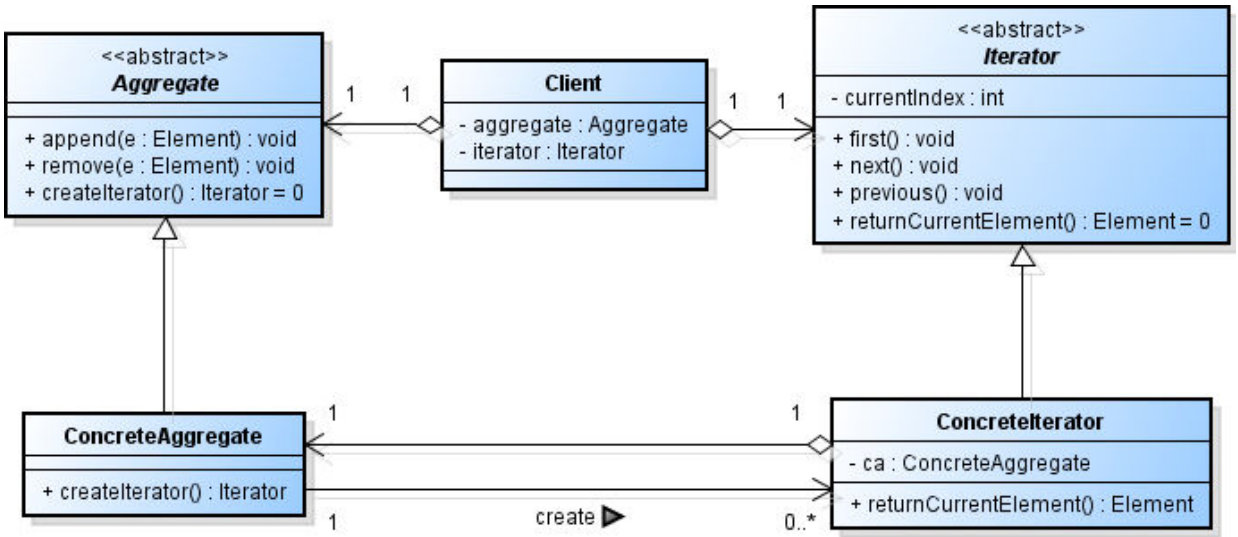


Figura 49 – Estrutura do padrão de projeto *Iterator*, adaptado de [GAMMA *et al.*, 1995]

Conforme apresenta a Figura 49, os participantes desse padrão são [GAMMA *et al.*, 1995]:

- ***Iterator***: define uma interface para acessar e percorrer elementos, mantendo a posição corrente do *Aggregate*;
- ***ConcreteIterator***: implementa a interface do iterador;
- ***Aggregate***: define uma interface para adicionar e remover elementos da lista, bem como para criar um objeto iterador;
- ***ConcreteAggregate***: implementa a interface *Aggregate* para retornar uma instância do *ConcreteIterator* apropriado;
- ***Client***: é configurado com instâncias de objetos *ConcreteAggregate* e ConcreteIterator, mantendo uma referência para suas respectivas classes bases.

A utilização desse padrão prove vantagens como: (a) transparecer a interface de iterações entre elementos sem a necessidade de conhecê-los de fato; (b) permite ao cliente iterar sobre os elementos de uma lista por diferentes maneiras sem a preocupação com a forma com que isso é realizado; (c) possibilita a criação de iteradores independentes para uma mesma estrutura de dados.

Outrossim, este padrão cumpre o princípio *OCP*, uma vez que possibilita a criação de novas estruturas de dados e seus respectivos iteradores específicos, mantendo a premissa desse princípio de possibilitar a extensão sem influenciar o código existente.

De modo a exemplificar o uso do padrão *Iterator*, a Figura 50 o apresenta aplicado no sistema de pedido de vendas sobre o método que realiza o cálculo do valor total de um pedido.

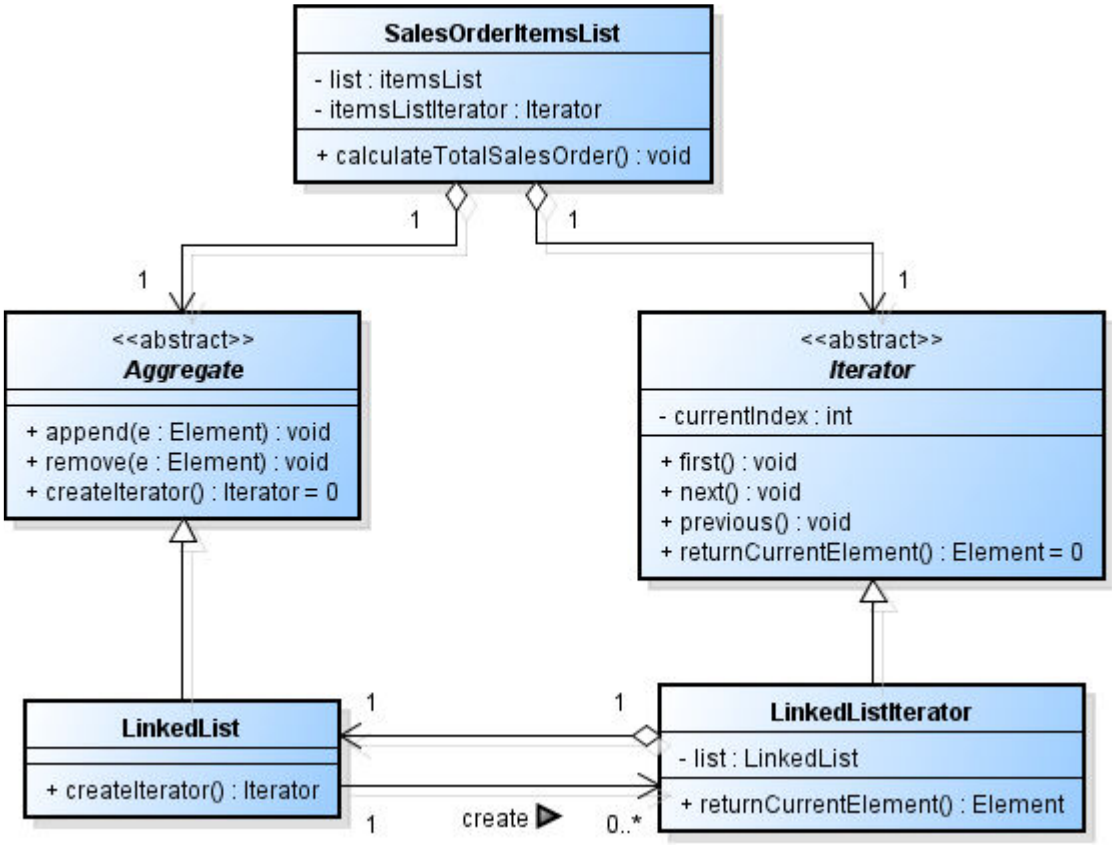


Figura 50 – Examplo do padrão *Iterator*

Conforme apresenta a Figura 50, o diagrama de classes ilustrado apresenta como cliente a classe *SalesOrderItemsList*, a qual é responsável por armazenar os itens de um pedido, bem como realizar o cálculo total do valor dos itens desse. O método *calculateTotalSalesOrder*, responsável pela soma dos valores individuais dos itens de um pedido, realiza o cálculo percorrendo e somando todos os itens armazenados na estrutura *LinkedList*, com o auxílio dos métodos presentes na classe *LinkedListIterator*.

Ademais, a separação da estrutura de dados do mecanismo de iteração permite com que o cliente defina iteradores para diferentes políticas de percorrimentos. No exemplo em questão, a classe *SalesOrderItemsList* poderia utilizar outro iterador para a realização de descontos individuais para cada um dos itens da lista, baseado em suas características. Ainda, essa separação assente ao princípio *SRP*, uma vez que a responsabilidade de gerenciamento de elementos e a responsabilidade de percorrimentos de tais elementos são tratadas por diferentes classes.

2.6.6 *Command*

O padrão comportamental *Command* permite que requisições sejam encapsuladas em forma de objetos, permitindo parametrizar outros objetos com diferentes solicitações, enfileirar ou registrar solicitações, e implementar recursos de cancelamento ou reversão de operações [FREEMAN *et al.*, 2004]. A Figura 51 apresenta a estrutura do padrão de projeto *Command*.

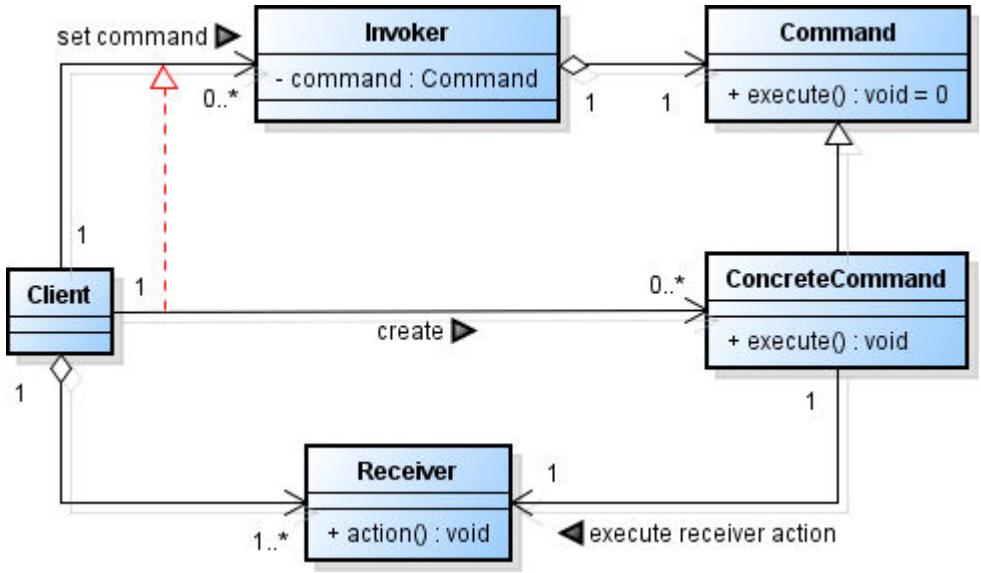


Figura 51 – Estrutura do padrão de projeto *Command*, adaptado de [GAMMA *et al.*, 1995]

De acordo com a Figura 51, a classe *Command* representa o centro do desacoplamento desse padrão e encapsula um *Receiver* com uma ação (ou conjunto de ações). A classe *Invoker* faz uma requisição a um objeto *Command* ao chamar seu método *execute*, o qual invoca as ações correspondentes no *Receiver*. Os participantes desse padrão são [GAMMA *et al.*, 1995]:

- ***Command***: declara uma interface para executar uma operação;
- ***ConcreteCommand***: implementa o método *execute* invocando o método *action* do *Receiver*;
- ***Client***: cria um objeto *ConcreteCommand* e seta seu *Receiver*;
- ***Invoker***: delega a requisição ao objeto *Command*;
- ***Receiver***: conhece a forma com que as requisições devem ser tratadas.

A utilização desse padrão prove vantagens como: (a) criar novos objetos de comando sem a necessidade de alterar o código existente; (b) desacopla os objetos que invocam operações dos quais as realizam; (c) possibilitar a parametrização dos clientes com comandos distintos, mesmo dinamicamente em tempo de execução.

Segundo Martin e Martin (2006), o padrão *Command* eleva o papel de uma função para o nível de uma classe. Por encapsular a noção de um comando, esse padrão permite desacoplar as interconexões lógicas de um sistema que normalmente seriam acopladas. Isso acontece pelo desacoplamento dos objetos que invocam uma operação dos quais realizam a tarefa de fato.

Ademais, o padrão *Command* possui uma característica interessante para a computação paralela/distribuída. O padrão fornece um meio de encapsular uma pequena parte da computação (*i.e.* um *Receiver* ou um conjunto de ações) em forma de um objeto, o qual pode ser invocado tempos depois. Na verdade, esse objeto pode ser invocado inclusive por uma *thread* diferente. Neste sentido, o padrão proporciona facilidades para criação de *schedulers*, *thread pools*, *job queues* etc. [FREEMAN *et al.*, 2004].

A classe responsável por tais escalonamentos de objetos *Command* é totalmente desacoplada dos objetos que estão executando a computação propriamente dita. Em um primeiro momento uma determina *thread* pode estar computando um cálculo aritmético e, no próximo, estar recuperando um pacote da rede. O escalonador de objetos só precisa se encarregar de enfileirar os comandos e chamar os métodos que realizam as funcionalidades encapsuladas no padrão [FREEMAN *et al.*, 2004].

Outrossim, o padrão *Command* possui algumas derivações úteis, como o caso do *NoCommand* [FREEMAN *et al.*, 2004] e o padrão *Active Object* [LAVENDER e SCHMIDT, 1996]. O padrão *NoCommand* é um exemplo de um objeto nulo. Um objeto nulo evita com que o cliente precise tratar de ponteiros nulos quando uma função não possuir um objeto concreto para retornar. Para isso, o cliente recebe um objeto concreto que, no entanto, não realiza nenhuma funcionalidade ao invocar seus métodos [FREEMAN *et al.*, 2004]. O padrão *Active Object*, por sua vez, é uma técnica antiga utilizada para implementação de múltiplas *threads* de controle, as quais tem sido utilizadas, de uma forma ou outra, para prover um núcleo simplificado de multitarefas para milhares de sistemas industriais [MARTIN e MARTIN, 2006].

2.6.7 *Composite*

O padrão estrutural *Composite* permite compor objetos em estruturas do tipo árvore para representar hierarquias todo-parte. Ademais, o padrão permite tratar objetos individuais e composições de objetos uniformemente [GAMMA *et al.*, 1995]. A estrutura do padrão permite com que ambos os objetos individuais e as composições processem as mesmas operações. Em outras palavras, na maioria dos casos é possível ignorar as diferenças entre as composições e os objetos individuais [FREEMAN *et al.*, 2004]. A Figura 52 apresenta a estrutura do padrão de projeto *Composite*.

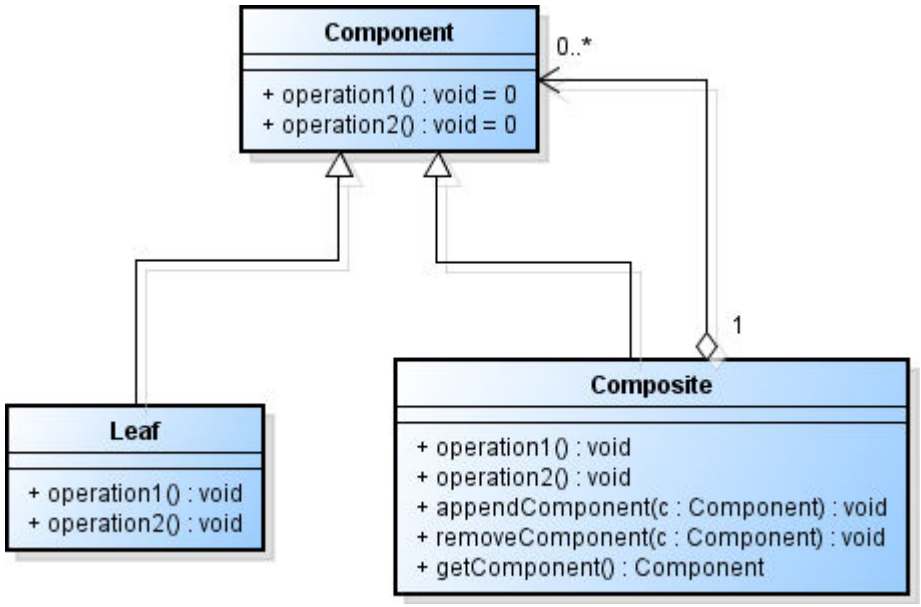


Figura 52 – Estrutura do padrão de projeto *Composite*, adaptado de [GAMMA *et al.*, 1995]

Conforme apresenta a Figura 52, os participantes desse padrão são [GAMMA *et al.*, 1995]:

- ***Component***: declara uma interface para os objetos colaboradores desse padrão;
- ***Leaf***: define as operações, representa os objetos folha da composição, onde esses não possuem filhos;
- ***Composite***: define as operações, armazena componentes filhos, e implementa operações relacionadas aos componentes filhos.

A utilização desse padrão prove vantagens como: (a) a possibilidade de tratar uniformemente objetos individuais ou compostos; (b) a facilidade de adicionar novas composições ou subclasses, uma vez que as estruturas existentes suportam quaisquer tipos de objetos derivados.

Outrossim, esse padrão é bastante utilizado na composição de interfaces para usuários, onde estruturas compostas como janelas, formulários e paneis, aglomeram entidades folhas como botões, campos de texto e barras de rolagem. Tais estruturas compostas permanecem presentes em algumas plataformas atuais de desenvolvimento, tais como a plataforma *Android* [ZECHNER, 2011]. Essa forma de compor interfaces para o usuário se mostra bastante flexível, uma vez que a criação de novos componentes não influencia no comportamento dos já existentes.

2.6.8 *Model View Controller*

O padrão composto ou arquitetural *Model View Controller* (*MVC*) foi inicialmente utilizado na construção de interfaces de usuário no Smalltalk-80 [KRASNER e POPE, 1988]. Esse padrão representa um conjunto de outros padrões de projeto que coexistem e colaboram em uma solução específica de um problema geral e recorrente. Os padrões que compõem essa solução são o padrão *Observer*, o padrão *Strategy* e, em alguns casos, o padrão *Composite* [GAMMA *et al.*, 1995]. Ademais, em algumas implementações, outros padrões podem ser aplicados como o *Facade*, na centralização do acesso à camada de dados, e o *Bridge*, na criação de interfaces multiplataformas.

Esse padrão consiste de três camadas, implementadas de maneira independente. O *Model* representa a camada de dados e o estado da aplicação; por sua vez, a *View* representa a camada de apresentação; por fim, o *Controller* define uma maneira com que a interface do usuário reaja à entrada de modo a requisitar alterações no *Model*. Antes da utilização desse padrão, os projetos de interface com o usuário tendiam a acoplar esses objetos em uma única estrutura. Sendo assim, o padrão MVC desacopla tais objetos provendo maior flexibilidade e reuso dessas partes [FREEMAN *et al.*, 2004].

Ademais, o padrão MVC desacopla a *View* e o *Model* ao estabelecer um protocolo de notificações entre eles. Uma *View* precisa assegurar que sua aparência

reflita o estado do *Model*. À medida que os dados do *Model* são alterados, esse notifica as *Views* que dependem dele. Em resposta, as *View*s interessadas podem se atualizar adequadamente a tais mudanças.

Essa abordagem permite anexar múltiplas *Views* ao *Model* de maneira a prover diferentes apresentações. E mais importante, novas *Views* podem ser criadas para um *Model* sem a necessidade de alterar o código existente. Esse mecanismo de notificações e desacoplamento entre entidades se dá através da aplicação do padrão *Observer* [GAMMA *et al., 1995*].

Outra característica do padrão MVC é que as *Views* podem ser aninhadas. A interface de um painel de controle, por exemplo, pode ser implementado com o empilhamento de um conjunto de componentes (*e.g.* botões, barras de rolagem, campos de entrada de dados *etc.*). Tal característica é implementada através do padrão *Composite* [GAMMA *et al., 1995*].

O padrão MVC também permite que mudanças em uma *View* respondam diferentemente a entrada de dados do usuário sem alterações em sua apresentação visual. Isso é possível com a utilização de múltiplos *Controllers*, os quais podem ser substituídos por outros de modo a apresentar comportamentos diferentes no tratamento de entrada de dados.

Em suma, uma *View* utiliza uma instância de um *Controller* de maneira a responder a entradas de dados de acordo com determinada estratégia. De maneira a implementar diferentes comportamentos, o padrão *Strategy* pode ser aplicado. Esse padrão é útil quando é desejável substituir o comportamento do *Controller* seja estaticamente ou dinamicamente por outro comportamento [GAMMA *et al., 1995*].

O objetivo dessa arquitetura é tornar a aplicação modular. A modularidade consiste em dividir a aplicação em componentes independentes. Isso facilita a resolução de problemas, geralmente transformando-os em problemas menores, os quais podem ser resolvidos mais facilmente. Ademais, o emprego do padrão MVC facilita o desenvolvimento e eventuais atualizações na aplicação, uma vez que os componentes são independentes.

A utilização desse padrão prove vantagens como: (a) desenvolver módulos de maneira individual; (b) realizar manutenções simplificadas; (c) proporciona facilidades em prover maior dinamismo na execução das aplicações. A Figura 53 apresenta a estrutura do padrão de projeto MVC e seus relacionamentos.

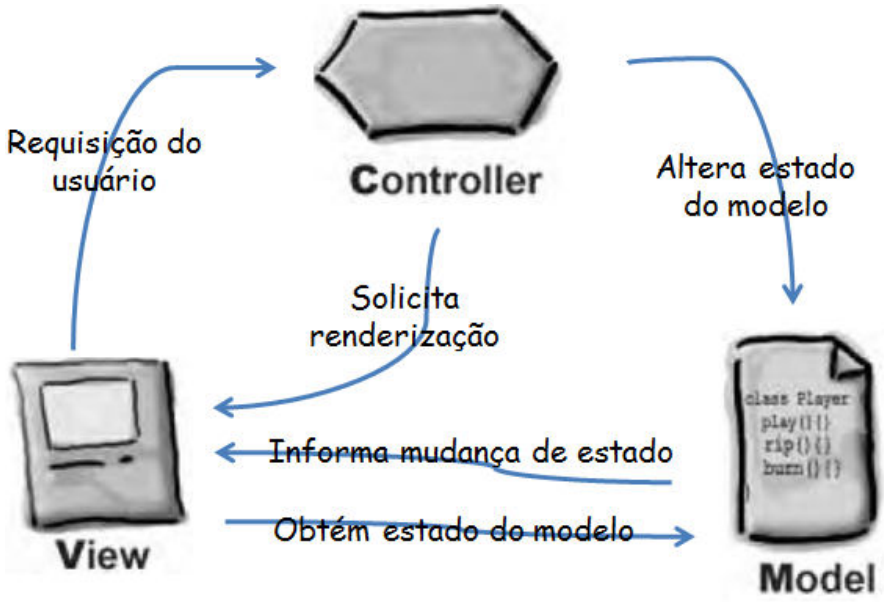


Figura 53 – Estrutura do padrão MVC, adaptado de [FREEMAN *et al.*, 2004]

Sucintamente, conforme ilustra a Figura 53, o *Controller* é responsável por interpretar as requisições do usuário realizada na *View* e manipular o *Model* baseado nessa entrada de dados. A cada alteração no estado do *Model*, esse notifica a *View* sobre o ocorrido. A *View*, por sua vez, obtém o estado atual do *Model* para se atualizar adequadamente. Ademais, o *Controller* solicita renderizações extras na *View* baseado em algumas regras de controle de interface da aplicação.

De modo a exemplificar o uso do padrão MVC, a Figura 54 o apresenta sob o viés dos demais padrões que compõem (ou podem compor) sua estrutura.

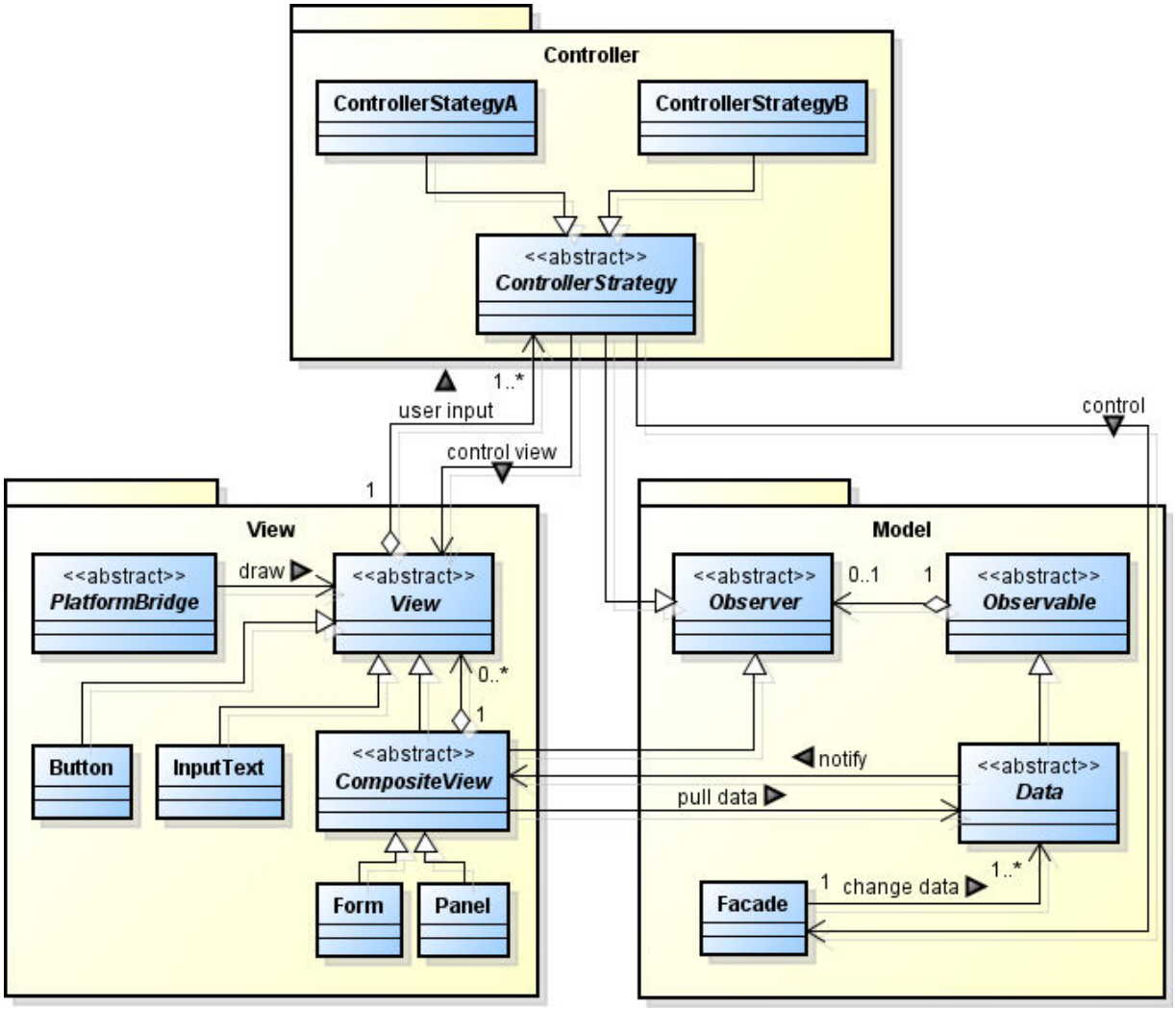


Figura 54 – Padrão MVC sob o viés dos padrões que compõem sua essência

De acordo com a Figura 54, o padrão *Composite* representa a interface com o usuário, consistindo de um conjunto aninhado de formulários, painéis, botões, campos de inserção de dados, entre outros. Cada componente da interface é um elemento composto (*e.g.* formulário), ou um elemento folha (*e.g.* botão). Ademais, na criação de interfaces, o padrão *Bridge* pode ser aplicado de modo a desenhar interfaces multiplataformas, onde cada implementação para determinada plataforma específica se encarrega da criação pontual de seus componentes.

O padrão *Strategy*, por sua vez, é representado na interação entre a *View* e o *Controller*, onde a *View* é configurada com uma dada estratégia provida pelo *Controller*. A *View* é responsável apenas pelos aspectos visuais da aplicação, e encarrega o *Controller* na tomada de decisões relacionada ao comportamento da interface.

Por fim, o *Model* implementa o padrão *Observer* de maneira a manter objetos interessados em seu estado atualizados. Ao utilizar o padrão *Observer*, o modelo se mantém completamente independente de suas *Views* e *Controllers*. Ademais, a comunicação entre o *Controller* e o *Model* pode ser simplificado com o padrão *Facade*, na centralização do acesso à camada de dados.

O alcance do mínimo acoplamento obtido através da correta implementação do padrão MVC permite que o *Model* reflita seu estado em múltiplas *Views*, sem necessariamente conhecer sobre suas implementações. Outrossim, o *Controller* separa a lógica operacional do *Model* apenas requisitando alterações pontuais em seu estado. Ao manter as três entidades do padrão minimamente acopladas, o projeto de uma dada aplicação oferece maior reusabilidade e extensibilidade.

2.7 PADRÕES DE IMPLEMENTAÇÃO

De maneira geral, padronizações possibilitam que desenvolvedores compartilhem de um mesmo vocabulário na estruturação de projetos de *software*. Uma vez desenvolvido esse vocabulário em comum, a comunicação entre desenvolvedores se torna mais fácil e inspira aqueles que não conhecem a linguagem a aprendê-la [FREEMAN *et al.*, 2004]. Isso também eleva o pensamento do desenvolvedor a pensar em termos de padrões e não apenas no simples encadeamento de linhas de código. Não obstante, a correta aplicação dos princípios dos padrões aumentam a flexibilidade e manutenibilidade do código.

Padrões de implementação ou de codificação, particularmente, promovem melhor qualidade ao *software* como um todo, reduzem o tempo de desenvolvimento, promovem trabalho em equipe e eliminam tempo desperdiçado com dificuldades para o entendimento da codificação existente, simplificando assim, sua manutenção [SUTTER e ALEXANDRESCU, 2004]. Além disso, com experiência na adoção de tais boas práticas, os desenvolvedores tendem a construir algoritmos mais eficientes em sua primeira versão, minimizando o tempo gasto com refatorações de código.

De acordo com McLaughlin *et al.* (2006), escrever *software* é como qualquer outra forma de escrita. Assim como a escrita de um artigo, os pensamentos devem ser rascunhados e refinados até a leitura desses se tornar agradável. A leitura de um código-fonte deveria ser como a leitura de um livro. Em um primeiro momento, o

leitor deveria ser capaz de dizer o que está acontecendo e, mesmo que ainda sobrem algumas dúvidas, a continuação da leitura deveria ajudar a encontrar as respostas para tais dúvidas. Ademais, bons desenvolvedores e projetistas deveriam estar sempre dispostos a gastar um tempo extra na escrita de um código legível, simplesmente pelo fato disso melhorar a habilidade de manter e reutilizar o mesmo [MCLAUGHLIN *et al.*, 2006].

Neste sentido, esta seção apresenta alguns dos padrões de codificação mais comuns e adequados para a composição de *software*[4]. Para isso, a Subseção 2.7.1 apresenta dicas para a nomeação dos elementos que compõem um *software*. A Subseção 2.7.2, particularmente, apresenta dicas para a composição de funções mais coesas, de maneira a evitar redundâncias no código. A Subseção 2.7.3, por sua vez, descreve as situações que a utilização de comentários no código pode ser uma má ideia, bem como as situações ideais para a presença desses. Por fim, a Subseção 2.7.4 descreve sucintamente boas práticas para a formatação adequada e organização da estrutura do código de uma aplicação.

2.7.1 Nomenclatura dos elementos de *software*

A nomenclatura adequada dos elementos que compõem *software* é algo de grande importância, uma vez que o *software* como um todo é composto por elementos que precisam ser nomeados. Em um *software* são nomeados os atributos ou variáveis, os métodos ou funções, as classes e suas respectivas instâncias, entre outros elementos que compõem sua estrutura [MARTIN, 2008]. Neste âmbito, a nomenclatura de tais elementos deveria seguir determinadas regras ou padrões de modo a facilitar o entendimento da codificação do *software* como um todo.

A seguir são apresentadas as regras mais usuais na nomenclatura de elementos em um *software*.

[4] As duas referências principais sobre padrões de implementação utilizadas neste trabalho (*i.e.* [BECK, 2007] e [MARTIN, 2008]), em geral descrevem as boas práticas de maneira similar, porém com nomenclaturas distintas. Neste sentido, este trabalho apresenta tais boas práticas sem nomeá-las de fato, descrevendo apenas suas características e aplicações.

A. Nomear os elementos de modo que esses revelem sua existência

O nome de uma variável, função ou classe deveria responder inicialmente o porquê de sua existência, o que faz e como é usado. Outrossim, aconselha-se escolher e adotar um único padrão para nomear classes que englobam um certo conceito abstrato, tais como *fecth*, *retrive* e *get* para recuperar o estado de um atributo [MARTIN, 2008]. O Algoritmo 3 apresenta um exemplo de duas nomenclaturas, uma pobre (linha 1) e outra rica em detalhes (linha 3).

```
1 double d; // Distance in meters.
2 . . .
3 double distanceInMeters;
```

Algoritmo 3 – Exemplo do uso de nomes significativos

O nome da variável *d* não revela de fato sua existência, nem o que faz e como é usada. Para compensar a falta de detalhes, desenvolvedores tendem a colocar comentários para amenizar o problema. Entretanto, apesar de amenizar o problema localmente, a utilização de tal variável ao longo do código tornará o mesmo ininteligível, especialmente em conjunto com outras variáveis pobremente nomeadas. A nomenclatura adequada nesse caso deveria especificar o que está sendo medido e a respectiva unidade de medida utilizada [MARTIN, 2008].

B. Nomear classes e objetos com substantivos

Classes e objetos deveriam ser nomeados com substantivos, tais como *Route* e *curitibaToManaus*, *Aviator* e *john* etc. Ademais, a distinção entre classes e suas respectivas instâncias é dada pelo uso da primeira letra; maiúscula para classes e minúscula para objetos. Ainda, nomes com palavras compostas deveriam seguir o padrão *CamelCase*, onde cada nova palavra é iniciada com letra maiúscula, unidas e sem espaços [BECK, 2007].

C. Nomear funções e métodos com verbos no infinitivo

Funções e métodos deveriam ser nomeados com verbos no infinitivo, tais como *acelerateTurbine*, *moveYokeToLeft*, *updateSpeed* etc., de modo que indiquem

uma ação a ser executada. Outrossim, de acordo com o padrão *javabean*, métodos que apenas resgatam ou definem um valor para um atributo específico, deveriam ser nomeados com seu respectivo nome, precedido dos prefixos *get*, *set* e *is*, como por exemplo, *getSpeed*, *setNumberOfPassangers* e *isAirplaneFull*. Tal padrão é altamente difundido e utilizado na Programação Orientada a Objetos em diversas linguagens.

2.7.2 Composição de funções e métodos

Nos primórdios da programação, uma das maiores dificuldades de desenvolvimento era a falta de modularização na composição de *software*. Tais programas eram estruturados, organizados em uma rotina gigante com controles de fluxo que saltavam de blocos em blocos (*i.e. goto*). Tais construções eram confusas, difíceis de ler e, principalmente, dificultavam a distinção das partes importantes das menos importantes. Além disso, devido ao alto acoplamento presente no código, os desenvolvedores tinham dificuldade em reutilizar tal código, o que implicava na reescrita de código redundante.

Atualmente na programação, a separação de código através de funções e métodos auxilia na composição de programas mais coesos, uma vez que tais estruturas viabilizam a reutilização de código. A primeira regra para a composição de funções coesas é a de que essas deveriam ser pequenas. Isso auxilia na redução ou até eliminação de código redundante. Um meio eficiente para determinar o tamanho de uma função/método é definir apenas uma funcionalidade para sua existência. O princípio de projeto *SRP*, descrito na Subseção 2.4.3, define especificamente essa regra, não só para a composição de funções em geral, mas sim para todos os elementos de um *software* [MARTIN, 2008].

Uma vez que a funcionalidade de uma função é bem-definida, sua nomenclatura se torna bastante obvia. Unindo a coesão de uma função com um nome significativo, proporciona maior legibilidade para o *software* como um todo. Neste sentido, o Algoritmo 4 ilustra um exemplo de um método bem-definido.

```
1  static const int CONVERSION_FACTOR_MPS_TO_KPH = 3.6;
2
3  double Speedometer::convertSpeedToKPH(double speedInMetersPerSecond) {
4    return (speedInMetersPerSecond * CONVERSION_FACTOR_MPS_TO_KPH);
5  }
```

Algoritmo 4 – Exemplo de um método coeso

A nomenclatura das variáveis e constantes que compõem o método *convertSpeedToKPH* da classe *Speedometer* apresentado no Algoritmo 4, dispensam comentários adicionais, uma vez que o objetivo de tal método é bem-definido e coeso. Apesar da implementação do método apresentado ser pequena, métodos maiores poderiam ser implementados sem maiores problemas, desde que suas responsabilidades reflitam diretamente em seus nomes. Além disso, tais métodos só deveriam realizar operações internas menores caso essas não sejam compartilhadas por outros métodos. Caso contrário, de maneira a evitar redundância na codificação, tais operações deveriam ser distribuídas em outros métodos menores.

2.7.3 Comentários significativos

Uma vez que as regras citadas nas subseções anteriores sejam cumpridas, a utilização de comentários acaba tornando a leitura do código redundante, o que faz com esses sejam dispensáveis na maioria dos casos. De acordo com Kernighan e Plaugher (1978), códigos mal estruturados não deveriam ser comentados com o objetivo de amenizar a dificuldade do entendimento do mesmo, mas sim reescritos de forma coesa e inteligível.

De acordo com Martin (2008), os comentários tendem a dificultar a leitura de um código, ao invés de auxiliar em tal atividade. Normalmente isso ocorre porque um trecho de código comentado é propenso a ser alterado e, com certa frequência, os comentários tendem a deixar de refletir tais mudanças. À medida que o comentário se deprecia e/ou se afasta do código que descreve, tal comentário se torna mais propenso a estar errado, o que de fato atrapalha o entendimento de um trecho de código.

Entretanto, existem casos em que comentários são necessários e/ou benéficos. Martin (2008) descreve alguns dos comentários enquadrados nessa categoria:

- **Comentários informativos**: comentário utilizado para facilitar a compreensão de métodos presentes em bibliotecas de terceiros, os quais muitas vezes podem utilizar parâmetros ou, até mesmo, padrões de entrada de dados com expressões regulares;
- **Comentários que justificam a intenção da existência de um elemento**: em alguns casos, na leitura de um bloco de código, podem aparecer elementos que a priori não apresentam um sentido muito claro. Neste âmbito, comentários que justifiquem a existência de tais elementos poderiam ser aplicados;
- **Avisos sobre consequências**: em alguns casos é útil avisar outros programadores sobre certas consequências. Por exemplo, um comentário sobre um método que possui elementos não-concorrentes, ou seja, que não deveriam ser executados em paralelo por outras *threads*;
- **Comentários *TODO***: algumas tarefas não terminadas ou a fazer poderiam receber um comentário *TODO*, explicando o que deveria ser implementado em tal trecho de código (*e.g.* método). Ademais, algumas ferramentas, como o Eclipse, fornecem uma interface que facilita a localização de comentários *TODO* ao longo do projeto.

2.7.4 Formatação e organização do código

A formatação do código escrito por uma equipe de desenvolvimento deveria seguir um conjunto específico e bem-definido de regras. Tais regras deveriam ser aplicadas consistentemente na construção de um *software*. Neste âmbito, a equipe de desenvolvimento deveria concordar com um único conjunto de regras de formatação e aplicá-lo invariavelmente. Além disso, uma vez definido tal conjunto, ferramentas automatizadas poderiam aplicar essas regras na formatação do código automaticamente [MARTIN, 2008].

Sucintamente, a formatação de um código deveria seguir um estilo de indentação única. Ademais, os desenvolvedores não deveriam economizar espaços entre elementos distintos, o que deixa a leitura do código mais clara.

Outrossim, a organização do código é um fator que também melhora a legibilidade de um código. Assim como a formatação adequada, a organização do código também poderia adotar algumas regras, tais como a ordem com que os elementos aparecem na estrutura de uma classe, por exemplo.

De modo a exemplificar um conjunto de regras de formatação e de organização, o Algoritmo 5 ilustra um trecho de código da classe *Aviator* do simulador de voo descrito na Subseção 2.4.3.

```
 1 #ifndef _AVIATOR_H_
 2 #define _AVIATOR_H_
 3
 4 #include "Behavior.h"
 5
 6 class Aviator {
 7
 8    public:
 9
10       Aviator(string name, Behavior *behavior);
11       virtual ~Aviator();
12
13    public:
14
15       Behavior *refBehavior;
16       Route *refRoute;
17       Airplane *refAirplane;
18
19    public:
20
21       void accelerateLeftEngine();
22       void accelerateRightEngine();
23
24       void pullStickToTakeOff();
25       void stabilizeFlight();
26       void pushStickToLandAirplane();
27
28 };
29
30 #endif
```

Algoritmo 5 – Formatação e organização do código da classe *Aviator*

Inicialmente, conforme ilustra o Algoritmo 5, todas as classes presentes no simulador de voo seguem um padrão de nomenclatura para os *defines*, conforme linhas 1 e 2. A seguir são incluídos todos os cabeçalhos de classes utilizados pela classe em questão. As linhas 6 à 28 compõem o bloco de definição da classe em questão.

É possível observar que existem espaços entre linhas em várias partes do trecho de código apresentado. Tais espaços facilitam a leitura do código, uma vez que mantem separadas partes distintas do mesmo.

Outrossim, nas linhas 8, 13 e 19, é possível observar a existência de três aberturas de blocos para definição de elementos públicos. Apesar do fato de que a presença das linhas 13 e 19 ser desnecessária, essa regra auxilia na separação dos elementos distintos de uma classe, como construtores e destrutores no primeiro bloco, referências para outras classes no segundo bloco, e definição de atributos e métodos no terceiro bloco.

Em suma, a formatação adequada, organizada e padronizada do código proporciona um aumento de legibilidade a esse, bem como simplifica a manutenção do *software* com um todo. Neste âmbito, os desenvolvedores tendem a minimizar o tempo gasto com questões de adequação da estrutura e formatação do código, permitindo com que esses se concentrem única e exclusivamente na elaboração da lógica do sistema.

2.8 CONCLUSÃO

Esse capítulo introduziu brevemente os princípais paradigmas de programação, de modo a complementar o trabalho realizado incialmente por Banaszewski [2009]. Ainda, o capítulo introduziu a base conceitual necessária para entender o estado da arte e da técnica em que se encontra o PON atualmente. Outrossim, o capítulo teve como foco principal o desenvolvimento de *software*, em especial nas questões de padronizações de projeto e de codificação.

Apesar de o PON se apresentar como uma nova solução, capaz de resolver parte dos problemas relacionados ao desempenho dos paradigmas de programação usuais, a qualidade do projeto e principalmente da codificação influencia diretamente no desempenho das aplicações desenvolvidas sob os princípios desse.

Neste âmbito, surge a proposta de aplicar padrões de projeto e de codificação na concepção de aplicações PON, atendendo necessidades como desempenho, facilidade de manutenção, legibilidade de código entre outros. Para isso, as demais seções apresentam boas práticas e técnicas de programação específicas para esse paradigma.

3 CONTRIBUIÇÕES PARA O PON E SUA MATERIALIZAÇÃO

Este capítulo contextualiza as contribuições deste trabalho para com o PON, bem como para a estrutura interna de seu *framework,* particularmente sob o viés de padrões de projeto. Neste âmbito, de forma assaz inovadora, a Seção 3.1 explica a essência do PON e também de seu *framework* sob o viés de padrões de projeto. A Seção 3.2, por sua vez, apresenta os novos conceitos introduzidos no âmago do *Framework* PON. A Seção 3.3, particularmente, apresenta as novas funcionalidades facilitadoras para a composição e depuração de aplicações PON. Por fim, a Seção 3.4 apresenta as conclusões deste capítulo.

3.1 PON E SEU NOVO *FRAMEWORK* SOB O VIÉS DE PADRÕES DE PROJETO

Esta seção apresenta primeiramente a compreensão da estrutura elementar do PON sob o viés de padrões de projeto e de implementação. Ainda, esta seção também apresenta a compreensão de materialização (*framework*) do PON, sob o viés de padrões de projeto. Por fim, relacionado a isso, a seção apresenta ainda as contribuições propostas neste trabalho que corroboraram para a concepção da nova materialização do PON.

O Arquétipo ou *Framework* PON, desde sua concepção prototipal (proposta por J. M. Simão), até a sua concepção dita primária ou original (proposta por R. F. Banaszewski), possibilitou a criação de aplicações regidas sob os princípios do PON [BANASZEWSKI, 2009]. Entretanto, a estrutura do *framework* original (e então vigente) apresentava algumas deficiências, tanto em questões de desempenho [VALENÇA *et al.*, 2011], quanto em questões de estrutura [SIMÃO *et al.*, 2012c]. Tais deficiências impossibilitavam o uso mais efetivo dos benefícios do PON e mesmo a criação de certos tipos de aplicações puramente desenvolvidas sob os princípios dele.

Neste âmbito, paralelamente e sinergicamente a este presente trabalho, outro trabalho relacionado (de mestrado) foi arquitetado com o foco em otimizações pontuais na estrutura de dados e algoritmos internos do *Framework* PON. Tais otimizações visam ganhos de desempenho, principalmente na estrutura de

notificações que formam a base desse paradigma [VALENÇA *et al.*, 2011; VALENÇA, 2012; SIMÃO *et al.*, 2012b; 2012c]. Assim sendo, o presente trabalho considera essa nova versão do *Framework* PON, o que se constitui no estado da técnica para a composição de *software* no PON, propondo e aplicando a sua estruturação por meio de padrões.

Neste sentido, a Subseção 3.1.1 considera o padrão *Observer*, que é fundamental para propor a compreensão (das notificações pontuais entre entidades) do PON em si sob o viés de padrões de projeto, bem como da compreensão do *Framework* PON sob este viés. A Subseção 3.1.2, por sua vez, considera o padrão *Iterator* complementar a estrutura de notificações. Ainda, a Subseção 3.1.3 considera o padrão *Abstract Factory* eficaz na criação de entidades PON sob o viés de múltiplas famílias de entidades, voltando-se mais ao *Framework* PON. A Subseção 3.1.4, particularmente, considera o padrão *Singleton* e o padrão *Strategy* complementares ao padrão *Abstract Factory* no âmbito da criação de entidades PON baseado em estruturas de dados distintas. Por fim, a Subseção 3.1.5 considera o padrão *Command* importante no processo de desacoplamento das chamadas de *Methods* (de entidades do PON, bem como de seu *framework*) realizadas por outras entidades.

3.1.1 *Observer*

Diferentemente da maneira passiva de inferência encontrada na maioria dos paradigmas usuais, o PON “evolui” as entidades, dotando-as com características reativas, o que permite com que essas reajam apenas a mudanças em seus estados lógicos via notificações/renotificações pontuais, evitando avaliações redundantes. No lugar de pesquisa e percorrimentos sobre entidades passivas (*e.g.* comandos e variáveis) ou semipassivas (*e.g.* módulos com alguma reatividade), ocorrem as notificações entre as entidades colaboradoras pertinentes [SIMÃO *et al.*, 2012a].

Neste sentido, cada relação baseada em notificações existente no PON pode ser entendida como um caso particular de aplicação do Padrão *Observer*, o qual foi explicado na Subseção 2.6.3. A utilização do padrão *Observer* em aplicações de naturezas distintas é vasta, geralmente aplicada na composição de entidades maiores (*e.g.* sistemas, agentes, módulos etc.). Entretanto, no PON e seu

framework, o conceito é utilizado em um nível mais “baixo” ou “atômico” aos comumente aplicados.

Na verdade, o conceito do padrão *Observer* é extrapolado no PON, uma vez que sua aplicação está presente na essência da inferência ou cálculo lógico-causal desse paradigma. Mais precisamente, o padrão *Observer* é extrapolado uma vez que está presente na aprovação ou desaprovação de cada entidade lógico-causal (*e.g.* “se-então”) em função dos valores ou estados de cada entidade factual (*e.g.* “atributo ou variável”) pertinente [SIMÃO *et al.*, 2012a]. Neste âmbito, o modelo conceitual do padrão *Observer* aplicado ao modelo do PON conforme ilustrado no diagrama de classes exposto na Figura 55, sendo que esse modelo se constitui no núcleo do *Framework* do PON.

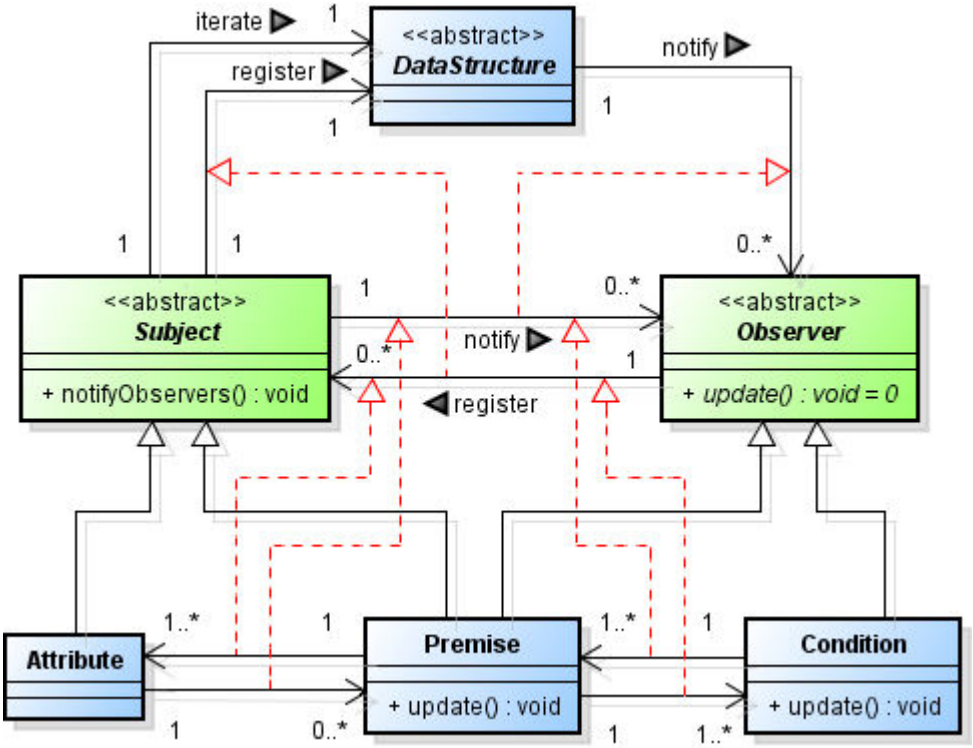


Figura 55 – Padrão *Observer* aplicado no processo de (re)notificações

Conforme ilustra a Figura 55, a estrutura do padrão *Observer* é composta por duas classes principais, isto é, a classe *Subject*, responsável pelas notificações pontuais aos objetos interessados em seu estado (*i.e.* observadores), e pela classe *Observer*, responsável pela execução de suas responsabilidades a cada notificação recebida do sujeito. Ademais, os observadores são arranjados em estruturas de dados (*i.e. DataStructure*) de maneira a facilitar as notificações pontuais por parte da

classe *Subject*. Neste âmbito, o Algoritmo 6 apresenta um exemplo de implementação do processo de notificações sob o viés do padrão *Observer*.

```
1  void Subject::notifyObservers() {
2
3     if (logicalValueChanged()) {
4
5        while (dataStructure->iterate()) {
6           dataStructure->getEntity()->update();
7        }
8
9     }
10
11 }
```

Algoritmo 6 – Exemplo de implementação do padrão *Observer*

Sucintamente, o Algoritmo 6 exemplifica uma implementação hipotética do processo de notificações. Nesse, a entidade notificadora, ao apresentar mudanças em seu estado lógico, percorre a lista de entidades observadoras, invocando seus respectivos métodos de atualização de estado.

De maneira geral, no *Framework* PON, as entidades que apresentam o comportamento de notificadoras são as instâncias das classes *Attribute* e *Premise*, enquanto as instâncias das classes *Premise* e *Condition* apresentam o comportamento de observadoras. Ademais, a entidade *Premise*, apresenta na verdade a definição de ambos os comportamentos. De fato, cada instância da classe *Premise*, ao receber notificações da entidade *Attribute*, comporta-se como observadora. Ainda, ao atualizar seu estado, quando conveniente, cada *Premise* atua notificando as entidades *Condition* interessadas na mudança em seu estado lógico.

Outrossim, as interpretações aqui vislumbradas poderiam apresentar diferentes versões caso novas soluções sejam propostas na estrutura do PON. Por exemplo, em [SIMÃO e STASZISZ, 2009b] e [SIMÃO *et al.*, 2010] foram propostas alterações no âmbito da cadeia de notificações, para fins de resolução de conflito e determinismo, o que consequentemente implicaria em uma nova interpretação do PON sob viés de padrões, particularmente no tocante aos diferentes usos do padrão *Observer*.

3.1.2 *Iterator*

De maneira geral, em termos práticos, cada entidade do PON depende de um meio para referenciar as entidades a serem notificadas. Na programação, cada entidade do PON depende de uma dada estrutura de dados (supostamente otimizada e apropriada) para armazenar tais referências. Ainda, o percorrimento de tal estrutura de dados se daria sem maiores conhecimentos de sua natureza interna de processamento, generalizando assim a solução para o tratamento de notificações em cada entidade do PON. Neste sentido, o padrão *Iterator*, dada suas características, poderia ser aplicado nesse sentido.

Neste âmbito, as deficiências relacionadas ao desempenho das aplicações desenvolvidas com o *Framework* PON original instigaram refatorações em seu código. Em tais refatorações, em especial no processo de notificações, a estrutura do *framework* precisou ser alterada, de maneira a possibilitar a criação de diferentes estruturas de dados e seus respectivos iteradores. Essa arquitetura possibilita a composição de diferentes estruturas de notificação entre as entidades PON que se adéquem às particularidades de diversos domínios de aplicações [VALENÇA *et al.*, 2011; VALENÇA, 2012].

Dentre as estruturas de dados disponíveis no atual *framework* se enquadram a estrutura de alto nível que implementa o contêiner *list* da *Standard Template Library* (*STL*) e três estruturas de dados implementadas especificamente para atender as necessidades do *framework*, nomeadamente *NOPLIST*, *NOPVECTOR* e *NOPHASH*. Cada nova estrutura implementada apresenta vantagens e desvantagens, que variam de acordo com as particularidades das aplicações [VALENÇA, 2012; SIMÃO *et al.*, 2012b; 2012c].

A implementação da estrutura de dados denominada *NOPLIST* valoriza a simplicidade de iteração sobre as entidades do PON. Isto se deve ao fato das iterações sobre as listas de entidades PON não necessitarem de operações como navegações bidirecionais, circulares ou particularmente operações de ordenações rebuscadas, como as encontradas em objetos da classe *STL* [VALENÇA, 2012].

A utilização da estrutura denominada *NOPVECTOR*, por sua vez, tem por objetivo realizar uma melhor utilização da *cache* de dados, ou seja, da memória principal, propriamente dita. Desta forma, o armazenamento dos elementos sobre a

estrutura *NOPVECTOR* é realizado de forma sequencial, através de um vetor de ponteiros para elementos PON. Neste âmbito, o acesso aos elementos PON é realizada também de maneira sequencial através da utilização de aritmética de ponteiros [VALENÇA, 2012; SIMÃO *et al.*, 2012b; 2012c].

Por sua vez, a estrutura *NOPHASH* foi implementada de maneira a realizar notificações pontuais, focando as entidades que apresentem o estado desejado igual ao estado atual da entidade em questão. Para isso, é necessário realizar de antemão o cálculo da função *hash* para direcionar as notificações apenas para as entidades interessadas. Ademais, as entidades que possuíam o estado verdadeiro, dito como anterior, são igualmente notificadas para terem seus estados lógicos reavaliados [VALENÇA, 2012].

Neste sentido, este trabalho propôs a aplicação do padrão de projeto *Iterator* no âmbito do percorrimento das estruturas de dados existentes. Para isso, o diagrama de classes exposto na Figura 56 apresenta o modelo conceitual do padrão *Iterator* aplicado ao *Framework* PON.

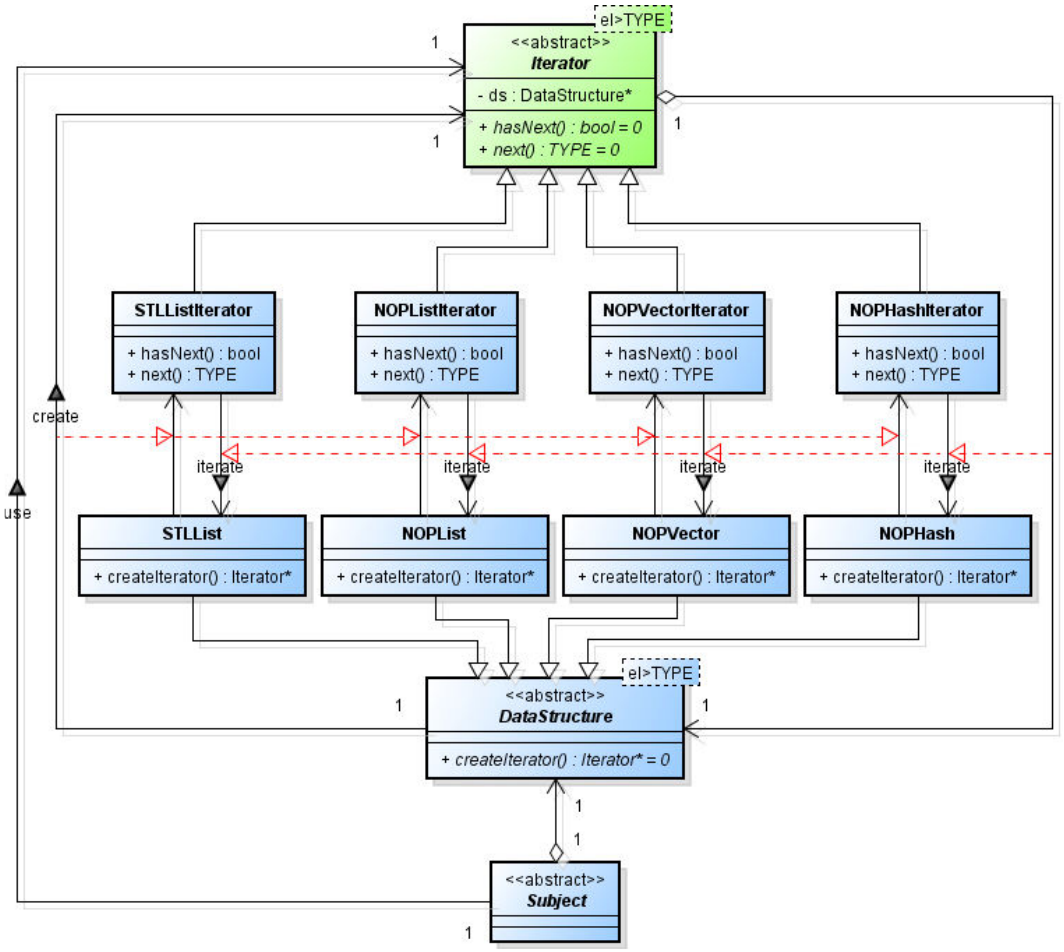


Figura 56 – Padrão *Iterator* aplicado no processo de iteração sobre entidades

Conforme ilustra o diagrama de classes da Figura 56, a classe principal *Iterator* é responsável por comportar uma estrutura de dados e fornecer métodos para acessar sequencialmente as referências das entidades armazenadas em sua estrutura. Ainda, a classe *DataStructure* é responsável por criar o iterador, passando uma instância dela mesma como referência para ele. Neste âmbito, o Algoritmo 7 apresenta um exemplo de implementação do percorrimento de entidades armazenadas em uma estrutura de dados do tipo lista encadeada sob o viés do padrão *Iterator*.

```
 1  Iterator *iterator = list::createIterator();
 2
 3  while (iterator->hasNext())
 4     iterator->next();
 5
 6  ----------------------------------------
 7
 8  TYPE NOPList::next() {
 9     pCurrent = pCurrent->pNext;
10     return pCurrent->element;
11  }
```

Algoritmo 7 – Exemplo de implementação do padrão *Iterator*

Conforme apresentado no Algoritmo 7, o ponteiro para um iterador é fornecido pela própria estrutura de dados, a qual este percorreria (linha 1). As linhas 3 e 4 demonstram o procedimento simplificado de iteração sobre uma dada estrutura de dados, no qual o iterador, ao verificar a existência de um próximo elemento (linha 3), concederia a possibilidade de retorná-lo (linha 4). Ademais, as linhas entre 8 e 11 exemplificam o procedimento de avançar para o próximo elemento da lista (linha 9), bem como retornar seu conteúdo para a instrução que o invocou (linha 10).

Outrossim, no *Framework* PON, as diversas estruturas de dados criadas apresentam particularidades distintas na forma com que armazenam e percorrem as referências para as entidades armazenadas. Entretanto, a utilização desse padrão possibilita com que todas as estruturas de dados sigam um modelo padronizado, facilitando a sua utilização (polimorficamente) na classe cliente. Mais precisamente, os métodos *hasNext* e *next* da classe *Iterator* são definidos como virtuais puros (*i.e.* abstratos), enquanto as classes concretas obrigatoriamente implementam as particularidades desses métodos de acordo com a estrutura de dados pertinente.

De acordo com Valença (2012), a estrutura *NOPVECTOR* apresenta o melhor desempenho na maioria dos casos, sendo adotada como estrutura padrão no

Framework PON. Em contra partida, quando a variação do estado de um dado *Attribute* é alta e, principalmente, a quantidade de *Premises* interessadas em seu estado for relativamente alta, a melhor estrutura para esse caso em particular é a utilização da estrutura *NOPHASH*[5].

3.1.3 *Abstract Factory*

A necessidade de criar diferentes estruturas de dados para ganhos de desempenho suscitou problemas relacionados à dificuldade de composição e instanciação de entidades PON. Tal problema é solucionado ao aplicar o padrão de projeto *Abstract Factory* na criação de tais entidades, uma vez que esse fornece uma interface abstrata e única para tal. A partir dessa interface, cada família de entidades (compostas por diferentes estruturas de dados) apresenta uma fábrica específica para a criação de tais entidades polimorficamente. De maneira a explicitar essa explicação, a Figura 57 ilustra a aplicação desse padrão na estrutura do *Framework* PON otimizado.

Conforme demonstra a Figura 57, a classe principal desde padrão é a fábrica abstrata (*i.e. EntitiesFactory*). Essa classe é responsável pela definição de todos os métodos abstratos responsáveis pela criação de entidades PON. O diagrama, em particular, abstrai a implementação real com a representação da classe abstrata *Entity*, a qual simboliza as entidades PON (*e.g. Attribute*, *Premise*, *Action etc*.).

[5] No trabalho de [VALENÇA, 2012] são apresentadas comparações e conclusões pontuais com maiores detalhes em relação à escolha da estrutura de dados para cada caso em particular.

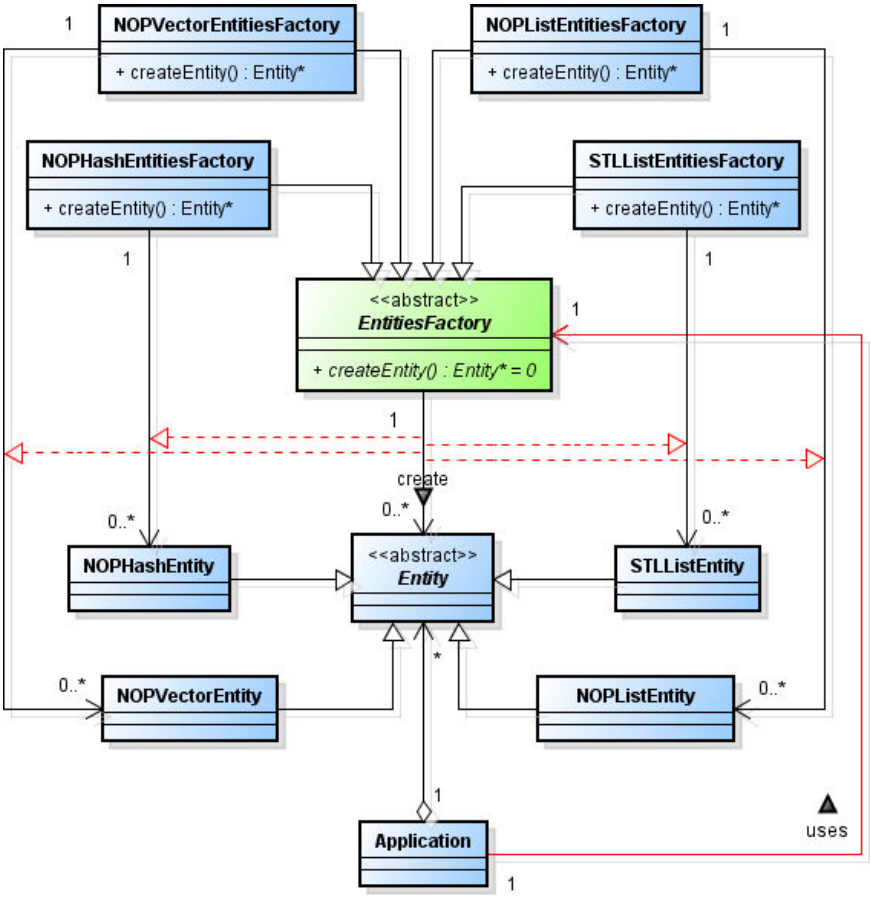


Figura 57 – Padrão *Abstract Factory* aplicado na criação de entidades

Outrossim, o diagrama ilustra a presença de outras fábricas, ditas concretas, que tem por finalidade implementar os métodos herdados da fábrica abstrata. Tais métodos têm por finalidade a criação de entidades PON concretas, as quais implementam internamente suas respectivas particularidades para cada estrutura de dados existente. De modo a elucidar a aplicação desse padrão na estrutura do *Framework* PON, o Algoritmo 8 exemplifica a sua implementação, bem como de sua utilização.

```
1  ### EntitiesFactory ###
2  virtual Boolean* createBoolean(bool value) = 0;
3
4  ### NOPVectorEntitiesFactory ###
5  Boolean* createBoolean(bool value) {
6    return new NOPVectorBoolean(value);
7  }
8
9  ---------------------------------------------------------------------
10
11 EntitiesFactory *entitiesFactory = new NOPVectorEntitiesFactory();
12 Boolean *atStatus = entitiesFactor->createBoolean(true);
```

Algoritmo 8 – Exemplo de implementação e utilização da *Abstract Factory*

De maneira geral, a estrutura da classe abstrata *EntitiesFactory* apresenta uma interface composta por métodos virtuais puros (*i.e.* abstratos). As fábricas concretas, por sua vez, precisam fornecer implementações para todos os métodos não implementados pela fábrica abstrata.

Neste sentido, conforme apresentado no Algoritmo 8, a linha 2 ilustra um exemplo de um método virtual puro para permitir a criação de *Attributes* do tipo *Boolean*. As linhas 5 a 7, por sua vez, apresentam a implementação efetiva de tal entidade composta por uma estrutura de dados do tipo *NOPVector*. Ainda, a linha 11, particularmente, demonstra o exemplo de criação de uma fábrica concreta do tipo *NOPVector*. Por fim, a linha 12 demonstra a utilização de tal fábrica para a criação de um *Attribute* do tipo *Boolean*. É possível observar que o cliente utiliza um ponteiro para uma fábrica de entidades abstrata, realizando chamadas polimórficas aos métodos de criação de entidades, que por sua vez são tratados pela instância da fábrica concreta *NOPVector*.

Particularmente, o padrão *Abstract Factory* flexibiliza a criação de entidades PON compostas por diferentes estruturas de dados, ao possibilitar a criação dessas com o uso de uma interface única (*i.e.* fábrica abstrata). Outrossim, tal padrão viabiliza a criação de novas famílias de entidades PON, compostas por eventuais novas estruturas de dados que possam vir a ser implementadas em trabalhos futuros.

3.1.4 *Singleton*

A criação facilitada de entidades PON propiciada pelo padrão de projeto *Abstract Factory* exige das classes que o utilizam apenas que possuam uma referência (ponteiro) para tal fábrica. Desta forma, tais classes recebem acesso aos métodos de criação fornecidos pela fábrica de entidades, podendo criar tais entidades sem se preocupar com a forma com que tais objetos são instanciados.

Neste âmbito, de modo a facilitar o acesso a tal fábrica, bem como garantir que exista apenas uma única instância de uma fábrica concreta em tempo de execução, o padrão de *software Singleton* foi implementado na estrutura do *Framework* PON. Ainda, o padrão garante que tal fábrica crie entidades PON uniformemente, de modo a se adequar a estrutura de dados definida.

A estrutura do padrão *Singleton* suportando a viabilidade do padrão *Abstract Factory* pode ser visualizada na Figura 58.

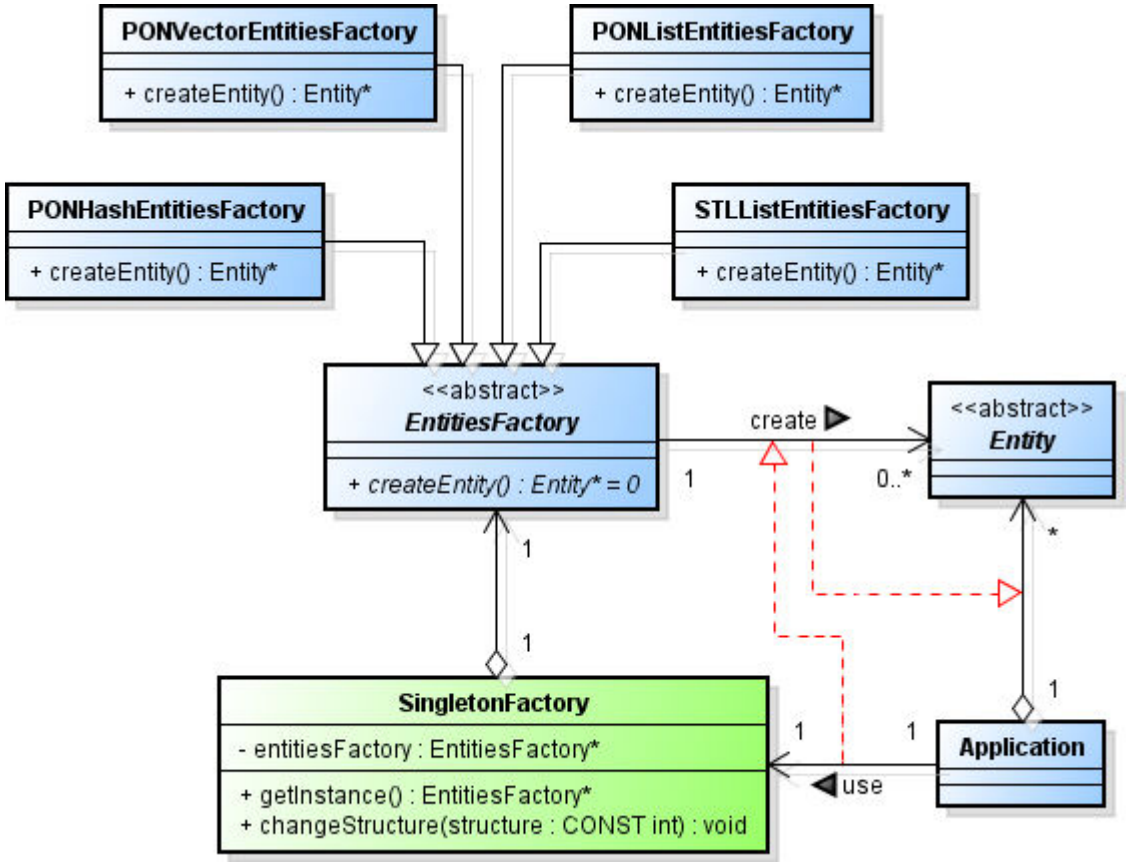


Figura 58 – Padrão *Singleton* aplicados no processo de criação de entidades

Conforme ilustra o diagrama de classes da Figura 58, a classe centralizadora *SingletonFactory* é responsável por manter uma referência para uma fábrica de entidades PON. Além disso, essa classe, ao seguir o padrão de *software Singleton*, assegura que apenas uma única instância de tal fábrica seja utilizada por vez na criação de entidades PON.

3.1.5 *Strategy*

Apesar de o padrão *Singleton* garantir a existência de apenas uma instância de uma fábrica de entidades PON, conforme dito anteriormente, há casos particulares em que entidades compostas por outra estrutura de dados poderiam apresentar maior eficiência em termos de custos de processamento.

Neste sentido, as entidades PON não precisariam necessariamente se adequar a uma estrutura de dados única, uma vez que tal estrutura é utilizada de maneira exclusiva por tais entidades. Esse fato se mostra favorável à composição de aplicações PON mais eficazes em questões de desempenho. Assim, isso viabilizaria a utilização de diferentes estruturas de dados em casos específicos, onde tais entidades desempenhem melhor sua função reativa (*i.e.* notificações pontuais).

Deste modo, o padrão de projeto *Strategy* foi implementado na arquitetura do *Framework* PON de modo a possibilitar a alteração da instância da fábrica em tempo de execução, permitindo assim a criação de entidades compostas por diferentes estruturas de dados. Outrossim, é importante ressaltar que esse padrão não inviabiliza a presença do padrão *Singleton*, uma vez que esse ainda atuaria no garantir da centralização da criação de entidades PON.

De modo a elucidar a aplicação dos padrões *Strategy* e *Singleton* na concepção de aplicações PON, o Algoritmo 9 exemplifica suas respectivas utilizações.

```
1  SingletonFactory::changeStructure(SingletonFactory::NOPVECTOR);
2
3  SingletonFactory::getInstance()->createBoolean(false);
4
5  SingletonFactory::changeStructure(SingletonFactory::NOPHASH);
6
7  SingletonFactory::getInstance()->createInteger(123);
```

Algoritmo 9 – Exemplo de utilização do padrão *Singleton* e *Strategy*

Conforme apresentado no Algoritmo 9, a linha 1 apresenta a maneira de definir a utilização de uma fábrica concreta do tipo *NOPVECTOR*. Esse procedimento seria necessário apenas na inicialização da aplicação, caso todas as entidades criadas possuíssem internamente a mesma estrutura de dados. A seu turno, a linha 3 representa um exemplo da organização propiciada pelo padrão *Singleton*, atuando na criação de uma entidade do tipo *Boolean*.

Ainda, conforme dito anteriormente, o padrão *Strategy* possibilita a alteração da instância para a fábrica de entidades em tempo de execução (linha 5). Ao passo que a estrutura é alterada, as criações subsequentes de entidades passam a adotar a nova estrutura definida. Nesse caso, a linha 7 representaria a criação de uma entidade do tipo *Integer* composta por uma estrutura de dados do tipo *NOPHASH*.

A união desses dois padrões em questão, juntamente com o padrão *Abstract Factory*, possibilita com que a classe responsável por armazenar uma referência para uma fábrica concreta possa receber referências de outras fábricas em tempo de execução, sem afetar de fato no fluxo de execução do programa. Com isso, entidades PON podem ser criadas deliberadamente sem influenciar no comportamento umas das outras, devido ao acoplamento mínimo propiciado pelo *framework*.

3.1.6 *Command*

O processo de notificações no *Framework* PON é constituído por todas as entidades que compõem a estrutura do PON definidos em sua teoria (cf. Figura 4) Entretanto, mesmo teoricamente, há distinções na maneira com que essas entidades se notificam.

Por um lado, as entidades responsáveis pelo cálculo lógico-causal apenas notificam as entidades interessadas em mudanças pontuais em seus estados lógicos. Tal mecanismo, conforme explicado anteriormente, foi concebido com a aplicação do padrão *Observer*. Por outro lado, as entidades responsáveis pela execução de uma *Rule*, quando aprovada, seguem um mecanismo de execução levemente diferente, notificando ou simplesmente instigando/comandando as demais entidades colaboradoras pertinentes para que executem determinada ação.

A Figura 59 ilustra o diagrama de classes que explicita a execução de uma *Rule* PON.

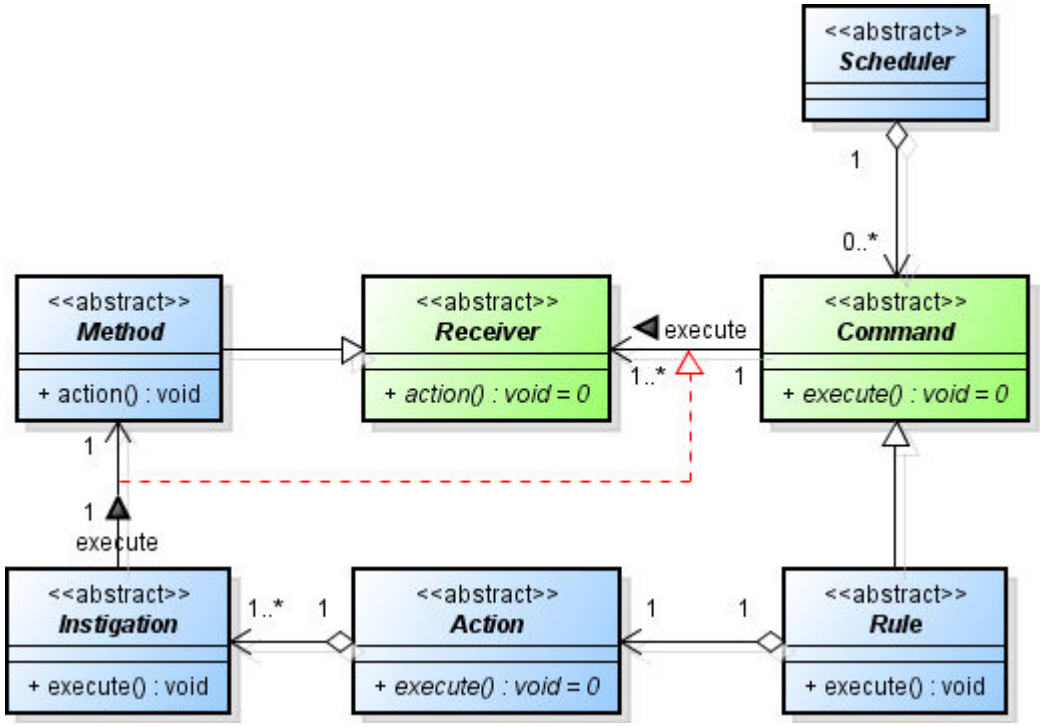


Figura 59 – Padrão *Command* aplicado na execução de *Rules*/*Methods* PON

Conforme ilustrado no diagrama de classes exposto na Figura 59, a aplicação do padrão de projeto *Command* no *Framework* PON apresenta duas classes principais. A classe *Command*, responsável pela ordem de execução de uma ação, tem uma relação indireta com a classe *Receiver*, a qual é responsável pela execução de tal ação. Neste caso, a classe *Rule* teria uma relação indireta com a classe *Method* por intermédio das demais classes colaboradoras (*i.e. Action* e *Instigation*).

Ademais, tais classes colaboradoras, cada qual com suas funções específicas, apresentam como uma de suas responsabilidades a delegação de execuções de uma determinada ação para as entidades apropriadas. Neste sentido, a classe *Rule* delegaria a execução para a classe *Action* que, por sua vez, delegaria a execução para a classe *Instigation*, que por fim instigaria a execução da classe *Method*. Como resultado final, cada *Rule* (*Command*) aprovada executaria a ação de um ou mais *Methods* (*Receiver*), possibilitando com que a execução desse(s) seja realizada de maneira independente.

```
1  void Instigation::execute() {
2
3     if (parametersList == 0) {
4        method->execute();
5     } else {
6        method->execute(parametersList);
7     }
8
9  }
```

Algoritmo 10 – Implementação do padrão *Command* no *Framework* PON

No exemplo ilustrado no Algoritmo 10, a entidade *Instigation* apresenta como responsabilidade principal a correta execução de uma entidade *Method.* Ademais, a entidade *Instigation*, quando considerado ambientes multiprocessados, seria responsável por tratar das peculiaridades desses ambientes, como a localização da entidade *Method* na distribuição proposta, tornando a presença do padrão de projeto *Command* ainda mais relevante no âmbito da execução de aplicações PON.

3.2 NOVOS CONCEITOS

Na seção anterior foi apresentado o PON e seu novo *framework* sob o viés de padrões de projeto. Na verdade, o *framework* foi implementado segundo este viés, o que permitiu a estruturação e desacoplamentos salientados. Outrossim, há outras contribuições desta dissertação para com o PON e seu *framework*. Justamente, esta seção apresenta novos conceitos propostos e implementados no âmbito do PON e de seu *framework*.

Tais conceitos focam em questões como ganho de desempenho, facilidades de desenvolvimento e purismo na implementação de aplicações PON. Para isso, a Subseção 3.2.1 contextualiza o conceito de *Attributes* ‘impertinentes’. A Subseção 3.2.2, por sua vez, apresenta o recurso de dependência entre *Rules*. Por fim, a Subseção 3.2.3 apresenta a proposta da utilização de *Methods* PON na composição de cálculos aritméticos.

3.2.1 *Attributes* ‘impertinentes’

De maneira geral, a reatividade presente nos *Attributes* proporcionaria uma execução livre de avaliações redundantes e desnecessárias, comuns aos paradigmas de programação usuais (cf. Seção 2.2). Entretanto, existem casos em que a variação de um *Attribute* encadearia sequências de notificações indesejáveis.

Isso ocorreria em situações onde um dado *Attribute* apresentaria constantes mudanças de estado, disparando o fluxo de notificações a cada variação, sem afetar efetivamente na aprovação da *Rule* a qual pertence. Assim, tais notificações desnecessárias impactariam negativamente no desempenho de execução de uma aplicação PON.

Por exemplo, pequenas mudanças em um *Attribute* referente à temperatura interna (*atTemperature*) de uma casa que seria pertinente a uma dada *Rule*. Esta *Rule* seria responsável por ativar o ar condicionado se a temperatura interna atingir um dado valor e se alguém estiver na casa (*atStatus true*). Entretanto, a casa passa maior parte do tempo vazia (*atStatus false*). Assim, a maior parte das notificações do *Attribute* em questão (*atTemperature*) seriam ‘impertinentes’.

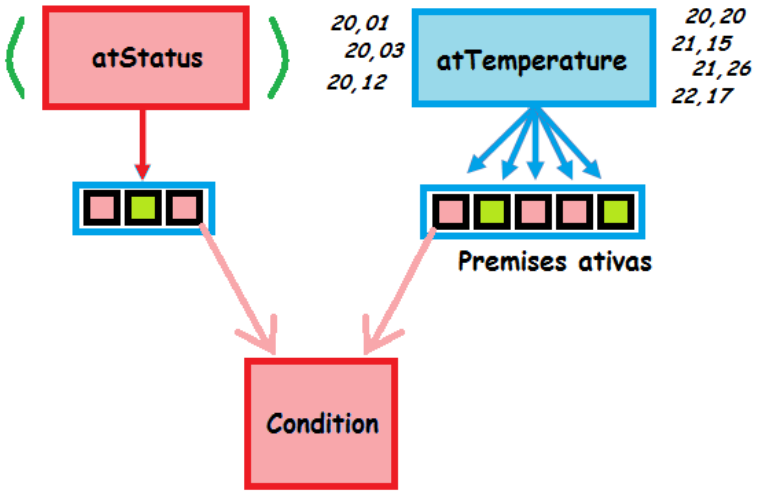


Figura 60 – Impacto nas alterações de estado de *Attributes* ativos

De modo a elucidar o problema em questão, considera-se o exemplo ilustrado na Figura 60. Neste exemplo são apresentados dois *Attributes* distintos, um do tipo *Boolean* (*atStatus*) e outro do tipo *Double* (*atTemperature*) em uma *Condition*/*Rule* composta por duas *Premises*. Uma *Premise* avaliaria se o estado de

atStatus é verdadeiro, enquanto a outra *Premise* avaliaria se o estado de *atTemperature* é maior do que um dado valor.

O *Attribute atStatus* apresentaria poucas mudanças em seu estado, permanecendo a maior parte do tempo com o estado *false*, disparando o fluxo de notificações esporadicamente. O *Attribute atTemperature*, por sua vez, apresentaria alterações constantes em seu estado, que no cenário em questão raramente impactariam na aprovação de sua *Condition*.

Neste sentido, um *Attribute* como *atTemperature* poderia ser categorizado como 'impertinente'. Sendo assim, de modo a evitar tal cenário de notificações inúteis, cada *Attribute* impertinente em dado contexto deveria ter suas funções reativas desabilitadas temporariamente para com as *Premises-Conditions-Rules* afetadas pela impertinência. Neste âmbito, a Figura 61 considera o mesmo cenário anterior, agora com a inativação temporária das *Premises* contendo *Attributes* 'impertinentes'.

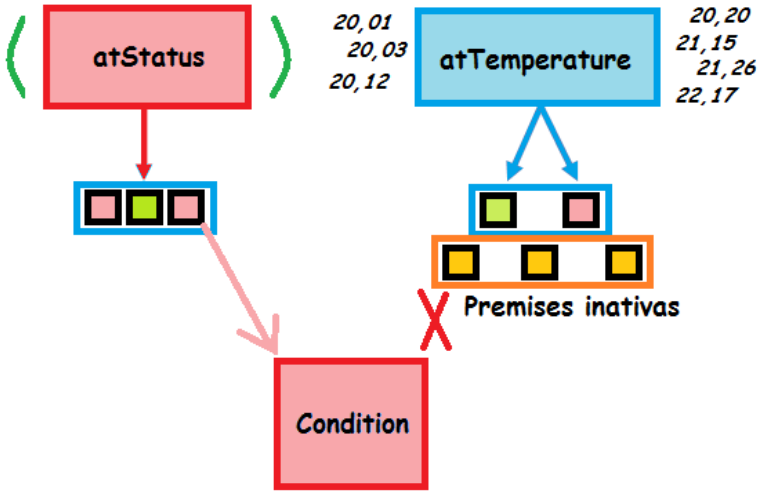


Figura 61 – Impacto nas alterações de estado de *Attributes* 'impertinentes'

Conforme apresenta o cenário ilustrado na Figura 61, ao inativar temporariamente algumas das *Premises* compostas pelo *Attribute* impertinente *atTemperature*, as variações de estado desse não impactariam no disparo do fluxo de notificações para tais entidades. Neste âmbito, quando o conjunto dos *Attributes* tivessem aprovado 'suas' *Premises* em uma dada *Condition-Rule*, essa deveria solicitar a reativação das notificações para a *Premise* composta pelo *Attribute* impertinente. Assim, uma vez que o *Attribute atStatus* apresentasse estado

verdadeiro, a *Condition-Rule* ilustrada solicitaria a reativação da *Premise* correspondente ao *Attribute atTemperature*, conforme ilustra a Figura 62.

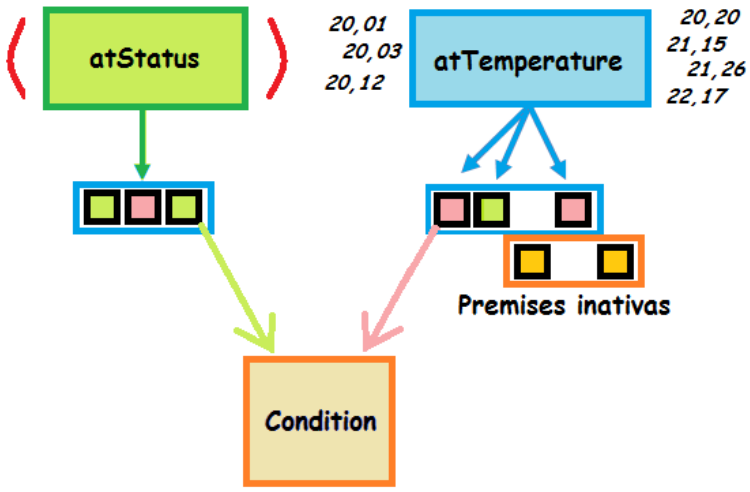


Figura 62 – Exemplo de reativação de uma entidade desativada

Neste sentido, conforme o cenário ilustrado na Figura 62, o *Attribute atStatus*, ao apresentar o estado verdadeiro, colocaria tal *Condition* em "ponto de aprovação", o que reativaria as funções reativas da *Premise* temporariamente desconsiderada. Assim, a entidade *Premise* em questão permitiria ser notificada (e mesmo demandaria notificações) novamente pelo *Attribute atTemperature*, atuando normalmente como uma entidade (re)ativa. Por fim, após a devida aprovação e execução da *Rule* em questão, o *Attribute* impertinente voltaria a ignorar tal *Premise* até que seja requisitado novamente por outra *Condition*.

De maneira geral, em termos práticos, esse conceito é implementado no F*ramework* PON por intermédio do próprio padrão de projeto *Observer*. De acordo com a apresentação desse padrão no âmbito das entidades PON (cf. Subseção 3.1.1), essas já apresentavam mecanismos para adicionar e remover entidades interessadas em mudanças em seu estado. Assim, entidades passíveis de impertinência atualmente fazem uso desses mecanismos em tempo de execução. Para melhor elucidar esse comportamento os algoritmos Algoritmo 11 e Algoritmo 12 demonstram o código desse conceito implementado no novo *Framework* PON.

```
1  if (amountImpertinentEntities > 0) {
2    if (impertinentEntitiesActive) {
3      if (amountEntitiesTrue < amountEntities)
4        this->deactivateImpertinentEntities();
5    } else {
6      if (amountEntitiesTrue == amountEntities)
7        this->activateImpertinentEntities();
8    }
9  }
```

Algoritmo 11 – Controle de ativação/desativação de entidades impertinentes

Conforme ilustra o Algoritmo 11, caso uma *Condition* apresentar entidades impertinentes (linha 1), essa verificaria inicialmente se tais entidades estariam ativas ou inativas (linhas 2 e 5). Em caso de tais entidades estarem ativas, a condição para desativar o recebimento de notificações (linha 4) seria o número de entidades aprovadas se apresentar menor do que o número total de entidades que compõem a *Condition* em questão (linha 3). Em caso de tais entidades estarem inativas, a condição para ativar o recebimento de notificações (linha 7) dependeria de todas as entidades normais estarem aprovadas (linha 6).

```
 1  void NOPVectorCondition::activateImpertinentEntities() {
 2    impertinentPremisesList.setFirst();
 3    while (impertinentPremisesList.hasNext())
 4      impertinentPremisesList.next()->addCondition(this);
 5  }
 6
 7  void NOPVectorCondition::deactivateImpertinentEntities() {
 8    impertinentPremisesList.setFirst();
 9    while (impertinentPremisesList.hasNext())
10      impertinentPremisesList.next()->removeCondition(this);
11  }
12
13  void NOPVectorPremise::addCondition(Condition *condition) {
14    conditionsList.addElement(condition);
15    if (conditionsList.getSize() == 1) {
16      this->attributeA->addPremise(this);
17      this->attributeB->addPremise(this);
18      this->logicValue = this->logicCalculus();
19    }
20  }
21
22  void NOPVectorPremise::removeCondition(Condition *condition) {
23    conditionsList.removeElement(condition);
24    if (conditionsList.getSize() == 0) {
25      this->attributeA->removePremise(this);
26      this->attributeB->removePremise(this);
27    }
28  }
```

Algoritmo 12 – Associações e desassociações entre entidades PON

Conforme ilustra o Algoritmo 12, tanto o método de ativação quanto o método de desativação de entidades impertinentes percorrem uma lista de *Premises* impertinentes, respectivamente adicionando (linhas 1 a 5) ou removendo (linhas 7 a 11) a *Condition* em questão da lista de notificações de tais entidades *Premise*. Ao adicionar uma *Condition* à lista de notificações de uma *Premise* (linhas 13 a 20), uma expressão condicional adicional verifica se tal *Condition* adicionada é a primeira e única nessa lista (linha 15). Em caso afirmativo, a entidade *Premise* se registra na lista de notificações dos *Attributes* pertinentes. Ao remover a única *Condition* da lista de notificações de uma *Premise* (linhas 22 a 28), a *Premise* em questão se desassocia dos *Attributes* temporariamente. Isso inclusive garante que *Attributes* só notifiquem *Premises* caso tenham pelo menos uma *Condition* associada a mudanças em seus estados.

Portanto, as entidades que demandam a desativação temporária de tais notificações se desregistrariam da 'lista de entidades observadoras', através do recurso de remoção, evitando assim o recebimento de notificações inúteis. Por outro lado, ao demandarem a reativação das notificações, tais entidades simplesmente se registrariam novamente na 'lista de entidades observadoras' das entidades em questão.

3.2.2 Dependência entre *Rules*

De maneira geral, existem casos que um conjunto considerável de *Rules* dependeria de *Premises* semelhantes para suas aprovações/execuções. A criação de entidades *Premise* únicas (com o mesmo teste lógico-causal) e seus respectivos compartilhamentos seriam a alternativa mais apropriada nesse cenário, evitando desta forma a presença de entidades redundantes. Entretanto, existem outros casos que tais *Rules* dependeriam de duas ou mais *Premises* compartilhadas, as quais notificariam todas as entidades interessadas em mudanças ocorridas em seu estado.

Neste sentido, notificações desnecessárias poderiam ser evitadas caso as *Premises* comuns a todas as *Rules* notificassem apenas uma *Rule*, ao invés de todas elas. Assim, no momento que todas as *Premises* dessa *Rule* única apresentassem estado verdadeiro, a ponto de aprovar a execução dessa, ela

notificaria as demais *Rules* interessadas. Neste âmbito, de modo a evitar tal sobrecarga de processamento e mesmo dificuldades de implementação (*i.e.* compartilhamento de entidades semelhantes e passividade a erros), a funcionalidade de *Rules* dependentes permite criar uma dependência entre *Rules* no PON, onde uma ou mais *Rules* dependeriam da execução de uma determinada *Rule* para, só então, executarem.

Neste caso, tais *Premises* fariam parte de uma única *Rule*, a qual faria o papel de *Rule* mestre. As demais *Rules* fariam um vínculo com essa *Rule*, de tal forma que dependeriam de sua aprovação e respectiva notificação para só então executarem. Essa dependência traria benefícios em questões de desempenho e facilidades na composição de aplicações.

No âmbito de eficiência na execução, tais *Rules* atuariam na redução de notificações geradas pelas *Premises* em questão, que direcionariam suas notificações apenas à *Rule* mestre. Em relação à facilidade de composição de aplicações, tal abordagem simplificaria a essência de todas as demais *Rules*, tornando o código mais legível e consequentemente mais manutenível. A Figura 63 exemplifica a estrutura da dependência de *Rules*.

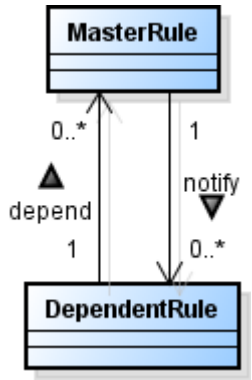


Figura 63 – Dependência entre *Rules*

Conforme ilustra a Figura 63, a dependência entre *Rules* é formada através de uma associação entre a classe dependente e a classe mestre. A classe mestre, por sua vez, notifica mudanças em seu estado para a(s) classe(s) dependente(s). É importante ressaltar que esse diagrama ilustra apenas conceitualmente a maneira como essa dependência é formada. No modelo/codificação real do *Framework* PON, ambas as funcionalidades de tais classes são definidas na classe *RulePlus*, conforme ilustra a Figura 64.

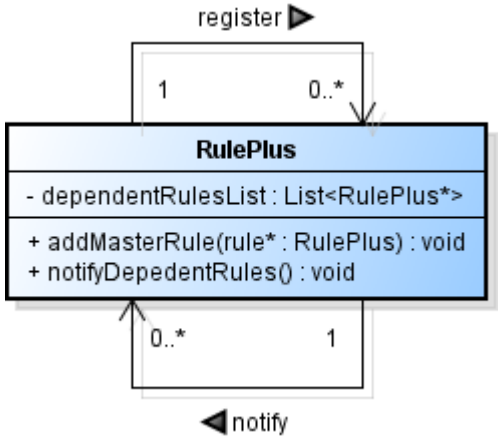


Figura 64 – Representação real da dependência entre *Rules*

Conforme ilustrado na Figura 64, a classe *RulePlus* apresenta em seu âmbito ambas as funcionalidades. Neste sentido, uma *RulePlus* poderia atuar tanto como uma *Rule* dependente quanto como uma *Rule* mestre, inclusive ao mesmo tempo. De maneira geral, em termos práticos, esse conceito apresenta as características do padrão de projeto *Observer*, sendo esse então aplicado na estrutura dessa classe.

Neste âmbito, a classe possibilitaria a adição de entidades interessadas em seu estado (*Rules* dependentes) que seriam notificadas a cada mudança de estado ocorrido na *Rule* mestre (*Rule* aprovada/desaprovada). De modo a facilitar o entendimento da utilização dessa funcionalidade, o Algoritmo 13 exemplifica o uso dela na composição de *Rules*.

```
1  RulePlus* rlAviatorCanOperate(Condition::CONJUNCTION);
2  rlAviatorCanOperate->addPremise(prAirplaneIsTurnedOn);
3  rlAviatorCanOperate->addPremise(prAirplaneHasNotReachedItsDestine);
4
5  RulePlus* rlAccelerateTurbines(Condition::CONJUNCTION);
6  rlAccelerateTurbines->addMasterRule(rlAviatorCanOperate);
7  rlAccelerateTurbines->addPremise(. . .);
8  rlAccelerateTurbines->addInstigation(. . .);
```

Algoritmo 13 – Exemplo de utilização da Dependência entre *Rules*

Conforme apresenta o trecho de código do Algoritmo 13, a segunda *Rule* utiliza o método *addMasterRule* (linha 6) e passa como referência a primeira *Rule* (*i.e. rlAviatorCanOperate*), com a finalidade de definir uma dependência entre a primeira e a segunda *Rule* do exemplo em questão. Acrescentando, a *Rule*

rlAccelerateTurbines representa uma *Rule* dependente, enquanto que a *Rule* *rlAviatorCanOperate* representa respectivamente a *Rule* mestre dessa.

Outrossim, ao ser criado um relacionamento de dependência entre duas *Rules*, a *Rule* dependente receberia internamente em sua estrutura um vínculo com uma *Premise* adicional, por meio de uma *SubCondition*. A mudança do estado lógico de tal *Premise* (provocado pela *Rule* mestre ao ser executada) faz com que o ciclo de notificações seja instigado. Deste modo, caso as demais *Premises* de uma *Rule* dependente apresentem estado lógico verdadeiro, a execução dessa *Rule* é ativada, dando prosseguimento a sua execução.

3.2.3 *Method* PON - Operações aritméticas

A estrutura do *Framework* PON original inviabilizava a criação de certos tipos de aplicações “puramente” desenvolvidas sob os princípios do PON, tais como as que envolvem operações aritméticas entre os *Attributes*. Tal fato exigia a criação de métodos do POO para a realização de tais operações e, inclusive, atribuições dos resultados em seus respectivos *Attributes*, o que de certa forma acrescenta complexidade adicional para a concepção de aplicações PON.

Além disso, as aplicações concebidas seguindo essa implementação apresentam problemas de acoplamento, uma vez que entidades desacopladas *Methods*, necessitam ser vinculadas a métodos do POO, definidos na estrutura de seus respectivos *FBEs*. Ainda, a composição de métodos do POO, conforme a Seção 2.1, possibilita a inserção de redundâncias em sua estrutura, o que de certa forma vai contra os princípios do PON.

Visando amenizar ou até mesmo eliminar esse problema, implementações específicas poderiam ser abstraídas do desenvolvedor por meio de módulos. Tais módulos atuariam na execução de uma funcionalidade específica, simplificando a interação das aplicações PON com recursos implementados sob os conceitos de outros paradigmas.

Neste sentido, este trabalho introduziu o uso do padrão *Composite* na implementação de operações aritméticas no *Framework* PON. Com tal funcionalidade, o desenvolvedor pode criar operações aritméticas de qualquer natureza e vinculá-las a um dado *Method*, que por sua vez executará o cálculo

dessa todas as vezes que a entidade for acionada. A título de exemplo a Figura 65 ilustra o diagrama de classes do padrão *Composite* aplicado no cálculo de operações aritméticas no PON.

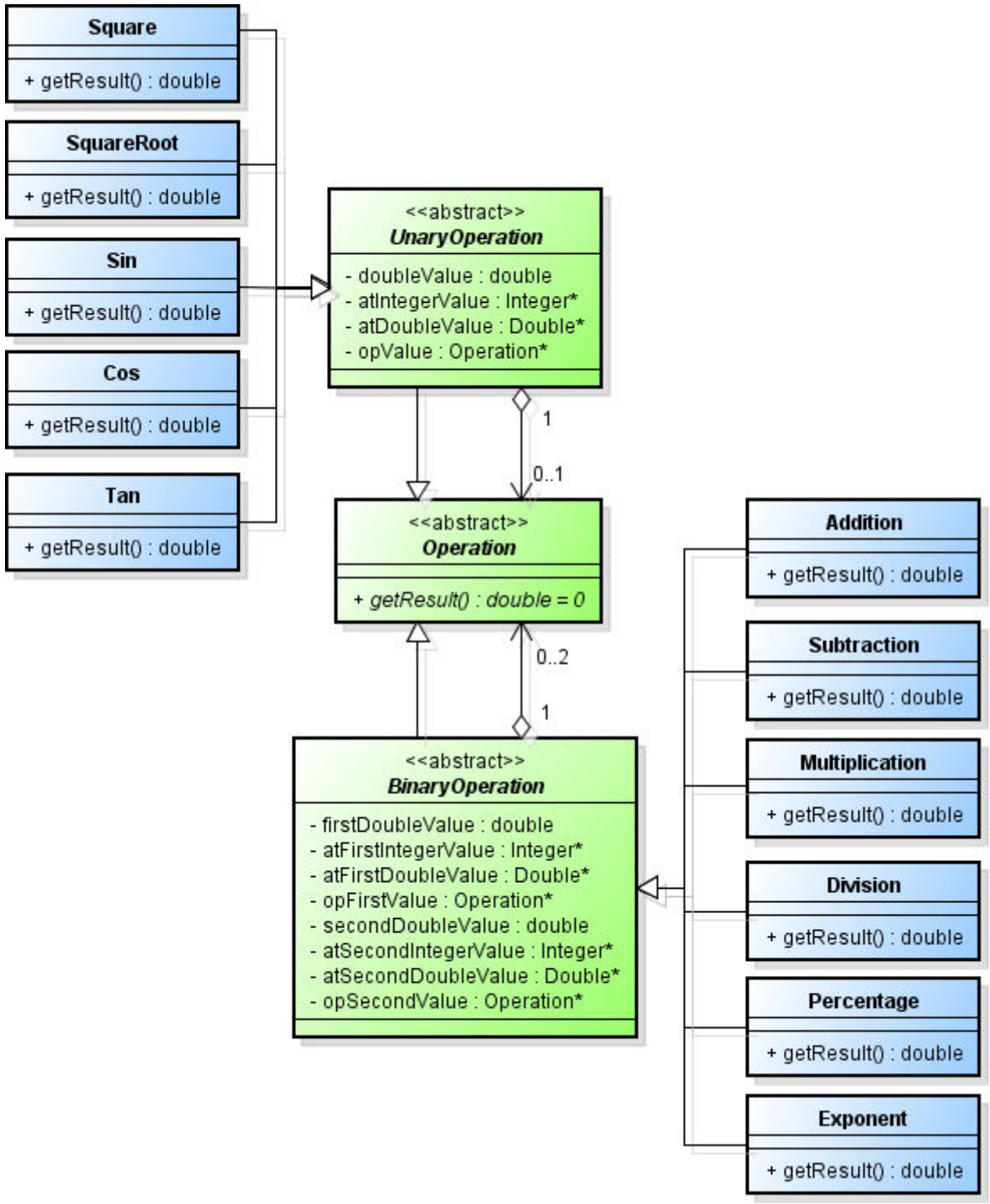


Figura 65 – Padrão *Composite* - Cálculo de operações aritméticas no PON

Conforme ilustrado no diagrama de classes exposto na Figura 65, as classes principais dessa solução são as classes bases *Operation*, *UnaryOperation* e

BinaryOperation. A classe *Operation*, particularmente, define um método virtual puro denominado *getResult*, com a finalidade de retornar o valor de sua respectiva operação. Ademais, tanto as operações unárias, quanto as binárias, ao herdarem da classe base *Operation*, implementam o retorno desse método, baseado em suas particularidades aritméticas.

A classe *UnaryOperation*, por sua vez, apresenta quatro atributos/objetos de tipos distintos. Entretanto, uma instância dessa classe é responsável por armazenar apenas um único valor, o qual é definido no construtor dessa classe. Tal valor pode ser uma variável primitiva, uma referência para um *Attribute* NOP numérico (*i.e. Integer* ou *Double*) ou uma referência para outra *Operation*. As classes *Square* e *SquareRoot* derivam da classe *UnaryOperation* e implementam o método *getResult*, realizando seus respectivos cálculos, com base no atributo definido em seus construtores.

A classe *BinaryOperation*, particularmente, realiza operações com base em dois valores. Tais valores, assim como com as operações unárias, podem ser do tipo primitivo, referências para *Attributes* numéricos ou referências para outras *Operations*. De modo a exemplificar a utilização do padrão *Composite* na estrutura do *Framework* PON otimizado, o Algoritmo 14 ilustra um exemplo de utilização de tal funcionalidade no estudo de caso sistemas de pedido de vendas.

```
1  Method *mtCalculateTotalValue =
2     SingletonFactory::getInstance()->createMethod(
3        salesOrderItem,
4        this->atTotalValue,
5        new Multiplication(salesOrderItem->atProduct->atPrice,
6                           salesOrderItem->atQuantity), 0);
```

Algoritmo 14 – Exemplo de utilização da funcionalidade *Method Operations*.

Conforme apresenta o Algoritmo 14, a composição de *Methods* PON teve sua essência simplificada, ao passo que evita com que redundâncias estruturais e temporais sejam inseridas na estrutura das aplicações PON. Na verdade, a implementação tradicional através de métodos do POO, sendo esses puramente aritméticos, não apresentariam tais problemas. Entretanto a nova abordagem garante que os *Methods* já existentes não possibilitem futuras alterações em sua estrutura que acarretem aos problemas descritos.

É importante ressaltar, entretanto, que a possibilidade de composição e vinculação de métodos do POO continua presente na estrutura do *framework*, e que

tal prática apresenta utilidades diversas. Dentre elas, citam-se a utilização de bibliotecas de terceiros (*e.g.* bibliotecas para acesso a bancos de dados e bibliotecas gráficas como a *OpenGL*) e possibilidade de inclusão de implementações de mais baixo nível em código *Assembly*.

3.3 NOVAS FUNCIONALIDADES

Esta seção apresenta duas novas funcionalidades implementadas no *Framework* PON. Tais funcionalidades visam principalmente simplificar o desenvolvimento e a manutenção de aplicações PON. Neste âmbito, a Subseção 3.3.1 apresenta facilitadores para a depuração de código PON. A Subseção 3.3.2, por sua vez, apresenta facilitadores para a composição de entidades PON.

3.3.1 Facilitadores para a depuração de código no *Framework* PON

A depuração de código no *Framework* PON é uma das dificuldades mais evidentes na concepção de aplicações PON. Mesmo com o uso de excelentes ferramentas de *debug* fornecidas pelas *IDEs* modernas existentes, acompanhar o fluxo de execução de um programa de natureza não tradicional pode se tornar uma tarefa árdua.

A atual implementação do *Framework* PON sob a linguagem de programação C++ define as cadeias de notificações a partir de estruturas de dados genéricas. Neste sentido, uma aplicação PON é baseada no conceito de percorrimento de tais estruturas, visto que esse mecanismo tem a responsabilidade de tratar das notificações pontuais para com as entidades referenciadas nessas. À medida que as ferramentas de *debug* percorrem sequencialmente as instruções de um programa, tais ferramentas tendem a adentrar em tais estruturas diversas vezes durante a execução de um programa. Assim, existia certa dificuldade em se saber exatamente sob qual dessas entidades a instrução corrente era executada.

Neste âmbito, uma ferramenta para facilitar o acompanhamento da execução de uma aplicação PON se faz pertinente e desejável. Para isso, algumas modificações na estrutura geral do *Framework* PON foram propostas e

implementadas. De maneira geral, as estruturas das classes de entidades PON sofreram algumas alterações, tais como estender a classe base *NOPEntity*, a qual basicamente fornece a funcionalidade de nomear tais entidades. Ademais, a classe *FBE* apresenta em sua estrutura a possibilidade de vincular um *FBE* com um *FBE* pai, chamado de *masterFBE*. A Figura 66 ilustra o diagrama de classes com as respectivas modificações.

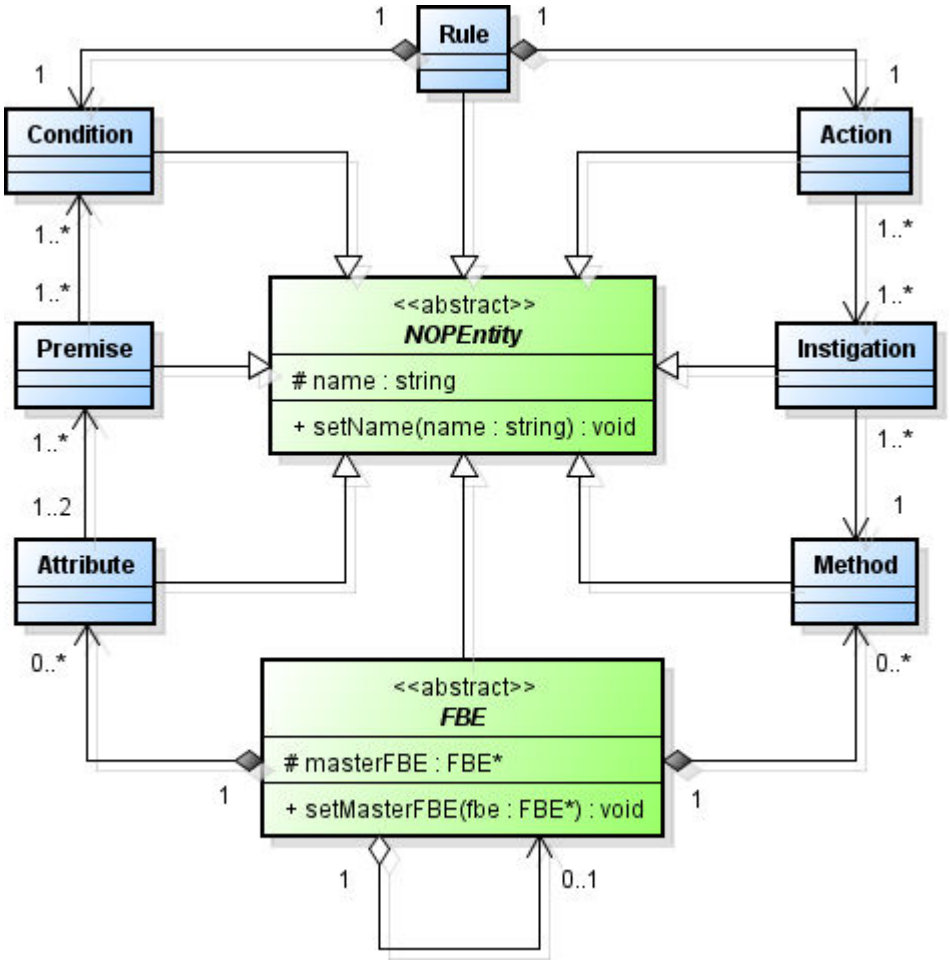


Figura 66 – Diagrama de classes referente à nova estrutura das entidades PON

Os recursos apresentados na Figura 66 proporcionam meios para que o desenvolvedor acompanhe diretamente no *debug* o nome de cada entidade PON, bem como possibilita determinar a precedência de cada *FBE* em relação a sua entidade pai (*master*). Isso facilita a depuração de código, tida como uma das dificuldades no âmbito da concepção de aplicações PON.

Ademais, tais recursos além de facilitar a utilização do *debug* tradicional, fazem parte da estrutura de uma ferramenta de *log* implementada no cerne do *Framework* PON, com o propósito de detalhar os passos de execução de uma

aplicação PON. Nesse âmbito, a Tabela 2 apresenta um trecho do *log* gerado na execução do simulador de voo.

Tabela 2 – Trecho do *log* gerado na execução do simulador de voo

```
1  + rlCheckStateAndTurnOnAirplane approved
2  * boeing->mtChangeToTakeOffFlightState executed
3  = boeing->atState = 2 (TAKE OFF)
4  + rlAviatorCanOperate approved
5  + rlFlapsSwitchInTakeOffFlightState approved
6  + rlSlatsSwitchInTakeOffFlightState approved
7  + rlStickMovedChangeElevator approved
8  * boeing->cockpit->flapsSwitch->mtChangeLeftFlapToTakeOffFlight executed
9  = boeing->LeftWing->Flap->atState = 2 (TAKE OFF)
10 * boeing->cockpit->flapsSwitch->mtChangeRightFlapToTakeOffFlight executed
11 = boeing->RightWing->Flap->atState = 2 (TAKE OFF)
12 - rlFlapsSwitchInNormalFlightState disapproved
13 * boeing->cockpit->slatsSwitch->mtChangeLeftSlatToTakeOffFlight executed
14 = boeing->LeftWing->Slat->atState = 2 (TAKE OFF)
15 * boeing->cockpit->slatsSwitch->mtChangeRightSlatToTakeOffFlight executed
16 = boeing->RightWing->Slat->atState = 2 (TAKE OFF)
17 - rlSlatsSwitchInNormalFlightState disapproved
18 * boeing->cockpit->stick->mtChangeLeftElevatorAngle
19 = boeing->Tail->LeftElevator->atAngle = 25
20 * boeing->cockpit->stick->mtChangeRightElevatorAngle
21 = boeing->Tail->RightElevator->atAngle = 25
22 - rlStickMovedChangeElevator disapproved
```

Conforme ilustrado na Tabela 2, o trecho de *log* gerado na execução do simulador de voo apresenta quatro recursos fundamentais, que são: (a) apresentar mudanças de estado de *Attributes*; (b) apresentar aprovações de *Rules*; (c) apresentar execuções de *Methods* e (d) apresentar desaprovações de *Rules*. Outrossim, tais recursos são também apresentados com símbolos, os quais facilitam suas distinções no *log*, onde atribuições são representados pelo símbolo (=), aprovações de *Rules* pelo símbolo (+), execuções de *Methods* pelo símbolo (*) e desaprovações de *Rules* pelo símbolo (-).

De maneira geral, é possível observar que a linha 1 apresenta uma aprovação de uma *Rule*, seguido de uma execução de um *Method* (linha 2) e posterior alteração de estado de um *Attribute* (linha 3), ambas realizadas pela execução da *Rule* aprovada. Assim, observa-se certa facilidade para o acompanhamento do fluxo de execução de uma aplicação PON, auxiliando de fato em sua composição e posteriores validações e manutenções.

Para utilizar tal funcionalidade, o desenvolvedor precisa apenas indicar o *stream* de saída de dados (*e.g. console*, arquivo texto etc.), conforme apresenta o código ilustrado no Algoritmo 15.

```
1  SingletonLog::changeStream(SingletonLog::CONSOLE);
2
3  SingletonLog::message("Exemplo de mensagem...");
4
5  SingletonLog::message("Exemplo de mensagem com valor: ", 123.45);
6
7  SingletonLog::changeStream(SingletonLog::NO_ONE);
```

Algoritmo 15 – Exemplo de utilização da funcionalidade de *Log*

É possível observar no código apresentado no Algoritmo 15 que as chamadas a métodos são centralizadas em uma classe baseada no padrão de projeto *Singleton*. Basicamente, ao ativar o tipo de *stream* desejado (linha 1 e 7), o *framework* apresentará os respectivos *logs* a medida com que o fluxo do programa seja executado. Ainda, a funcionalidade de *logs* permite com que o desenvolvedor insira suas próprias mensagens através do método *message* (linhas 3 e 5).

3.3.2 Facilitadores para a composição de entidades PON

A criação de entidades no *Framework* PON original se dava pela simples instanciação de uma classe do tipo desejado. Entretanto, com o surgimento de diferentes estruturas de dados no *Framework* PON otimizado (cf. descrito na Subseção 3.1.2), as entidades PON passaram a ser criadas com o auxílio de uma fábrica de entidades. Neste âmbito, o Algoritmo 16 exemplifica a criação de entidades PON com o auxilio da fábrica de entidades.

```
1  SingletonFactory::changeStructure(SingletonFactory::NOPVECTOR);
2
3  Premise* prPacmanHasNoMoreLives =
4     SingletonFactory::createPremise(
5        game->score->atNumLives, 0, Premise::EQUAL);
6  prPacmanHasNoMoreLives->setName("prPacmanHasNoMoreLives");
7
8  Method* mtChangeGameStateToGameOver =
9     SingletonFactory::createMethod(
10       game->atState, Game::GAME_OVER, Attribute::STANDARD);
11 mtChangeGameStateToGameOver->setName("mtChangeGameStateToGameOver");
12
13 RulePlus* rlCheckGameOverAndFinish =
14    SingletonFactory::createRulePlus("rlCheckGameOverAndFinish",
15       scheduler, Condition::CONJUNCTION);
16 rlCheckGameOverAndFinish->addPremise(prPacmanHasNoMoreLives);
17 rlCheckGameOverAndFinish->addInstigation(mtChangeGameStateToGameOver);
```

Algoritmo 16 – Exemplo de criação de entidades PON

Conforme ilustrado no Algoritmo 16, a criação de entidades PON se dá por meio da utilização de uma instância única (*Singleton*) da fábrica de entidades. A estrutura de dados para cada entidade pode ser definida em tempo de criação, pela chamada do método *changeStructure* (cf. linha 1).

Ademais, observa-se a utilização da fábrica para criação de uma *Premise* (linhas 4 e 5), de um *Method* (linhas 9 e 10) e de uma *Rule* (linhas 14 e 15). Ainda, de modo a facilitar a depuração do código, tais entidades são nomeadas com o próprio nome do ponteiro definido no código do programa, respectivamente nas linhas 6, 11 e 14.

Apesar dessas facilidades, é possível observar que as instruções utilizadas para tais criações não simplificam essa atividade (*i.e.* instruções longas e repetitivas). Outrossim, a própria nomeação de entidades é passível de erros, uma vez que dependeria do desenvolvedor adequar o nome do ponteiro do código com a nomenclatura da entidade em questão.

Neste sentido, de modo a proporcionar de certa forma um padrão de implementação específico para o PON, este trabalho propõe a utilização de pseudônimos (*i.e.* instruções *define* do C/C++) para simplificar a criação de entidades, bem como garantir sua integridade. Pseudônimos além de simplificar a utilização de uma instrução, possuem a capacidade de realizar um conjunto dessas com apenas uma única chamada. Basicamente, tais pseudônimos foram criados para abstrair chamadas para um ou mais métodos responsáveis pela criação de entidades PON, bem como para a definição de suas particularidades.

A título de exemplo, o Algoritmo 17 apresenta as definições dos pseudônimos para os tipos de *Attributes* PON. No código ilustrado são apresentados apenas os tipos *Boolean* e *Integer*, porém tal recurso é aplicado igualmente a todos os demais tipos de *Attributes*.

```
 1 #define BOOLEAN(fbe, attribute, value)
 2    attribute =
 3       SingletonFactory::getInstance()->createBoolean(fbe, value);
 4    attribute->setName(#attribute)
 5
 6 . . .
 7
 8 #define INTEGER(fbe, attribute, value)
 9    attribute =
10       SingletonFactory::getInstance()->createInteger(fbe, value);
11    attribute->setName(#attribute)
```

Algoritmo 17 – Definições de pseudônimos para os tipos de *Attributes* PON

Conforme apresenta o Algoritmo 17, a definição dos tipos de *Attributes* PON criam as entidades a partir de métodos presentes na fábrica de entidades instanciada (linhas 3 e 10) e posteriormente definem o nome de tal *Attribute* baseando-se no conjunto de caracteres utilizado para representar o ponteiro desse no código-fonte (linhas 4 e 11).

Igualmente a criação de entidades *Attribute*, as entidades *Method* também apresentam pseudônimos para suas respectivas criações e subsequentes definições. O Algoritmo 18 apresenta as definições dos pseudônimos para os tipos de *Methods* PON. No código ilustrado são apresentados três definições distintas, onde cada uma dessas possui uma instrução para criação de uma instância de um *Method* PON e outra instrução para definição de seu respectivo nome.

```
1  #define METHOD(fbe, method, attribute, value, flag)
2     method =
3        SingletonFactory::getInstance()->createMethod(
4           fbe, attribute, value, flag);
5     method->setName(#method)
6
7  #define METHOD_PLUS(fbe, method, attribute, value, operation, flag)
8     method =
9        SingletonFactory::getInstance()->createMethod(
10          fbe, attribute, value, operation, flag);
11    method->setName(#method)
12
13 #define METHOD_OPERATION(fbe, method, attribute, operation, flag)
14    method =
15       SingletonFactory::getInstance()->createMethod(
16          fbe, attribute, operation, flag);
17    method->setName(#method)
```

Algoritmo 18 – Definições de pseudônimos para os tipos de *Methods* PON

Outrossim, as diferenças entre os pseudônimos apresentados no Algoritmo 18 estão presentes em seus respectivos parâmetros de criação. A entidade *Method* criada no primeiro pseudônimo (linha 1) possui a função de apenas atribuir um dado valor (*value*) à um dado *Attribute*. A entidade *Method* criada no segundo pseudônimo (linha 7), por sua vez, possui a função de realizar uma operação (*operation*) aritmética simples (*i.e.* adição, subtração, multiplicação ou divisão) entre o *Attribute* e o valor (*value*), atribuindo o resultado de tal operação no respectivo *attribute.* Por fim, a entidade *Method* criada no terceiro pseudônimo (linha 13) possui a função de realizar uma expressão aritmética composta (*i.e.* funcionalidade *Operation* apresentada na Subseção 3.2.3), atribuindo o resultado dessa no *Attribute* definido na chamada desse pseudônimo.

Outrossim, as demais entidades PON (*i.e. Premise*, *Condition*, *Rule*, *Action* e *Instigation*) também apresentam pseudônimos, porém suas composições são similares as demais definições apresentadas. O Algoritmo 19 apresenta o mesmo exemplo de código demonstrado no Algoritmo 16, utilizando as simplificações proporcionadas pelos pseudônimos.

```
1  CHANGE_STRUCTURE(SingletonFactory::NOPVECTOR);
2
3  PREMISE(prPacmanHasNoMoreLives,
4     game->score->atNumLives, Premise::EQUAL);
5
6  METHOD(mtChangeGameStateToGameOver,
7     game->atState, Game::GAME_OVER, Attribute::STANDARD);
8
9  RULE(rlCheckGameOverAndFinish, scheduler, Condition::CONJUNCTION);
10 rlCheckGameOverAndFinish->addPremise(prPacmanHasNoMoreLives);
11 rlCheckGameOverAndFinish->addInstigation(mtChangeGameStateToGameOver);
```

Algoritmo 19 – Exemplo de criação de entidades PON utilizando os pseudônimos

É possível observar que a aplicação dos pseudônimos na criação de entidades PON simplifica suas composições, evita redundâncias nas chamadas de código e evita problemas com a integridade das mesmas (*i.e.* nomes e ponteiros com conjunto de caracteres divergentes).

Outrossim, é importante ressaltar que o uso de pseudônimos apresenta um custo adicional na 'montagem' de uma aplicação PON, uma vez que realiza algumas chamadas de métodos adicionais. Entretanto, esse custo de processamento é relativamente baixo, sendo praticamente imperceptível na composição da maioria dos domínios de aplicações. Na verdade, na execução de uma aplicação propriamente dita, esse recurso não representa nenhum impacto no desempenho, uma vez que durante a execução de uma aplicação, caso nenhuma entidade adicional seja criada, tal recurso não é utilizado.

3.4 CONCLUSÃO

Esse capítulo apresentou a essência do PON e de seu novo *framework*, bem como as particularidades desses sob uma nova perspectiva de leitura do PON, sob o viés de padrões de projeto. Tal perspectiva inicia pelo vislumbre da essência do paradigma e de seu *framework* sob a luz do padrão de projeto *Observer*.

De acordo com Simão *et al.* (2012), a solução de inferência do PON, responsável por determinar o fluxo de execução de uma aplicação desenvolvida sob os princípios desse, não se trata apenas de uma aplicação do conhecido padrão de projeto *Observer*. Tal solução é dita como uma extrapolação desse padrão, aplicada no âmbito do cálculo lógico-causal desse paradigma, através de notificações pontuais a pequenas entidades reativas.

Quanto ao *Framework* PON, de maneira geral, a sua estrutura apresentava em sua materialização original, os padrões *Iterator* e *Command*, além do padrão *Observer*. Tais padrões apesar de não estarem detalhados de forma explícita nos documentos referenciados neste trabalho, mantinham breves citações aos mesmos.

Neste presente trabalho, entretanto o *framework* recebeu melhorias significativas na aplicação de tais padrões, além de adicionar outros padrões de projeto em sua estrutura. Os padrões aplicados na estrutura do novo *Framework* PON tiveram como foco principal a simplificação do desenvolvimento de aplicações PON, proporcionada pela flexibilidade advinda da implementação desses. Dentre eles destacam-se o padrão *Abstract Factory*, *Singleton* e *Strategy*.

Ainda, nesse capítulo foram apresentados novos conceitos específicos para o PON, materializados na versão atual do *framework* em questão. Os conceitos de impertinência de *Attributes* e dependência entre *Rules* se adequám ao paradigma, visto que tratam de melhorias pontuais em sua estrutura, independentes de sua materialização.

Ademais, o conceito de operações aritméticas abstraídas em uma espécie de *Method* PON representa um conceito pontual para o *Framework* PON, não se encaixando explicitamente no estado da arte do paradigma. Tal conceito representa um módulo que elimina a necessidade do desenvolvedor criar métodos do POO para vincular em suas aplicações, tornando a implementação de aplicações PON, com operações aritméticas, mais “pura”. Ainda, outros recursos de programação como persistência em arquivos ou banco de dados poderiam ser implementados seguindo a mesma estrutura apresentada. Assim, os desenvolvedores focariam apenas no desenvolvimento de suas aplicações PON “puras”, relacionando as execuções de suas aplicações aos recursos preestabelecidos pelo *framework.*

Outrossim, a nova estrutura do *framework* apresenta novas funcionalidades que visam principalmente simplificar a composição e depuração de aplicações PON. Tais funcionalidades atuaram pontualmente no problema de acompanhar o fluxo de

execução não tradicional de aplicações PON por meio de depuradores tradicionais específicos para a Programação Imperativa.

Neste sentido, além de tais contribuições para a estrutura do *Framework* PON, vislumbra-se a possibilidade de explorar novos padrões de desenvolvimento específicos para o PON, analisando a influência que esses podem causar às aplicações, tanto no impacto em seus desempenhos, quanto em questões de praticidade de desenvolvimento.

Para melhores detalhes quanto à implementação adequada e eficiente de aplicações PON, o Capítulo 4 demonstra o uso efetivo da maioria das funcionalidades/recursos (antigas e novas) nos casos de estudo simulador de jogo *Pacman*, simulador de voo simples e sistema de pedido de vendas.

4 BOAS PRÁTICAS E PADRÕES DE DESENVOLVIMENTO PARA O PON

Os conceitos do chamado Paradigma Orientado a Notificações (PON) foram primeiramente utilizados no âmbito de aplicações de controle discreto para uma a composição de sistemas de manufatura simulados [SIMÃO, 2001; 2005; SIMÃO e STADZISZ, 2002; 2008; 2009a; 2009b; SIMÃO, STADZISZ e KÜNZLE, 2003; SIMÃO, STADZISZ e TACLA, 2009; SIMÃO *et al.,* 2010]. Em um dado período de tempo, a solução evoluiu de uma teoria de controle e inferência para um paradigma de programação [SIMÃO e STADZISZ, 2008; 2009a; SIMÃO *et al.*, 2012a]. No domínio da concepção de *software* nesse paradigma, as raízes do âmago da programação baseada na versão prototipal do *framework* e suas posteriores evoluções levaram a possibilidade de conceber *software* de várias naturezas sob os princípios desse paradigma.

Ainda, recentemente surgiu um método de desenvolvimento para o PON denominado Desenvolvimento Orientado a Notificações (DON) o qual permite elaborar artefatos de projeto voltados particularmente para a Programação Orientada a Notificações segundo um processo que se utiliza do conhecido Projeto Unificado, do ferramental da *Unified Modeling Language* (*UML*) e das pertinentes Redes de Petri [WIECHETECK, 2011; WIECHETECK *et al.*, 2011].

Entretanto, os trabalhos relativos ao DON e ao PON em geral não consideraram efetivamente questões de padrões de desenvolvimento de *software*. Além disso, a falta de um conjunto efetivo de programas exemplo, especialmente com escopos consideráveis, não contribuía efetivamente para um aprendizado orientado a exemplos.

De maneira geral, a implementação de aplicações PON apresenta certas complicações (*i.e.* fluxo de execução não convencional e dificuldades para depuração de código) que, em um primeiro momento, podem tornar sua concepção dificultosa. Neste sentido, este capítulo apresenta boas práticas e técnicas de implementação específicas para o PON, as quais são aqui entendidas como Padrões de Desenvolvimento para o PON. A proposta de tais padrões no atual estado da técnica do *Framework* PON se mostra bastante pertinente, uma vez que esses auxiliam os desenvolvedores a alcançar maior entendimento na programação baseada no PON como um todo.

Neste sentido, esta seção busca aliar as qualidades existentes no PON a um estilo de programação eficiente, dotado de boas práticas e padrões de desenvolvimento de *software*. Isso se dá especialmente na apresentação e implementação de programas de natureza mais realista (*e.g.* aplicações comerciais e simuladores), norteando o desenvolvimento de futuras novas aplicações. Outrossim, tal abordagem visa principalmente minimizar a curva de aprendizado desse novo estilo de programação. Ademais, tal estilo contribuiria para a elaboração de aplicações PON mais eficientes, melhor estruturados, modulares (com unidade coesa e desacopladas) e que possam ser estendidos e reutilizados com certa facilidade.

Outrossim, neste capítulo são apresentados alguns experimentos comparativos em termos de desempenho. Todas as execuções desses experimentos foram realizadas em um *notebook* com processador *AMD Turion X2* de 2.0 Ghz, com 3 GB de memória RAM. De maneira a evitar resultados imprecisos com sistemas operacionais que apresentam maior número de preempções, os testes foram executados em um sistema operacional Linux (distribuição *Debian*[6]). Ademais, o ambiente Linux oferece medidas de tempo mais precisas e, portanto, a unidade de medida apresentada nos resultados é representada em milissegundos. Outrossim, os resultados representam a média de execução de 100 repetições para cada experimento.

Para isso, este capítulo está organizado da seguinte maneira: A Seção 4.1 descreve detalhadamente os procedimentos para a criação de uma aplicação PON. Ainda, quando pertinente, sugere o uso de boas práticas e padrões de desenvolvimento elencados para a concepção de aplicações PON. Ademais, tal seção apresenta algumas considerações de implementação, comparando diferentes formas de implementação para problemas similares, considerando questões como desempenho, facilidade de composição de programas e purismo no estilo de programação (*i.e.* adotar uma implementação puramente baseada no PON, minimizando o uso de conceitos de outros paradigmas). A Seção 4.2, por sua vez,

[6] A distribuição *Debian* apresenta uma versão composta apenas pelos pacotes essenciais para a execução do sistema operacional, possuindo poucas aplicações que executam concorrentemente [DEBIAN, 2012]. Neste sentido, as aplicações tendem a executar sem muitas interferências externas (*i.e.* preempções) geradas por outras aplicações. Tal fato reforça a confiabilidade dos resultados obtidos em testes de desempenho.

demonstra a utilização pontual das boas práticas e padrões de desenvolvimento propostos em um caso de estudo específico. Mais precisamente, tal caso de estudo consiste em uma nova implementação do sistema de pedido de vendas apresentado na Subseção 2.4.2, objetivando justamente uma abordagem purista para com o uso dos conceitos do PON. Por fim, a Seção 4.3 apresenta as conclusões deste capítulo.

4.1 COMPOSIÇÃO DE APLICAÇÕES PON

A composição de aplicações PON baseada no novo *Framework* PON apresenta particularidades que o distinguem das demais implementações existentes. As particularidades dessa versão atual do *framework* estão descritas no Capítulo 3. Neste sentido, para desenvolver aplicações com o *Framework* PON em questão, inicialmente é necessário que o desenvolvedor estenda a classe *NOPApplication.* Tal classe proporciona uma ponte entre a aplicação PON e o cerne do *Framework* PON. A Figura 67 ilustra o diagrama de classes que representa o procedimento inicial de criação de uma aplicação PON.

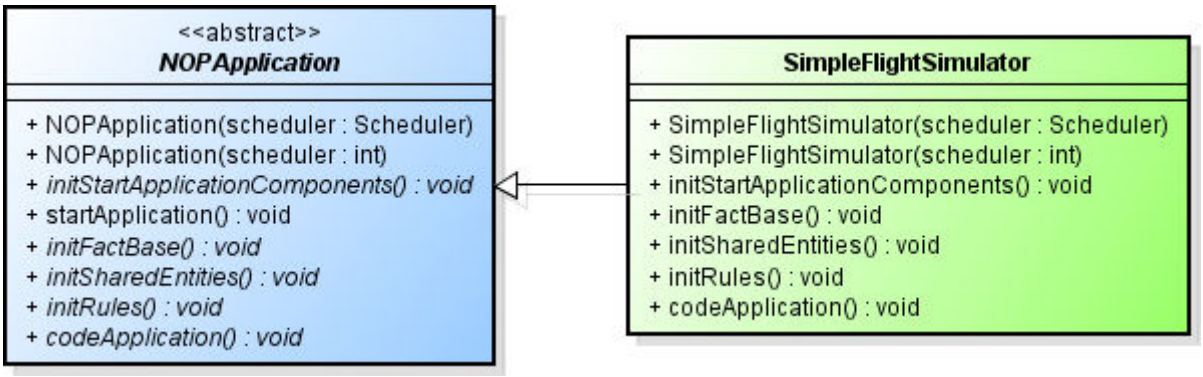


Figura 67 – Diagrama de classes do procedimento inicial de uma aplicação PON

Conforme apresenta a Figura 67, a classe *NOPApplication* apresenta métodos virtuais puros (*i.e.* abstratos) que devem ser implementados pela classe responsável pela inicialização de uma aplicação PON (*i.e. SimpleFlightSimulator*). Em tais métodos, o desenvolvedor deve se concentrar na inicialização e criação de *FBEs* e entidades compartilhadas, bem como na concepção de *Rules*. As etapas de configurações iniciais e implementação de uma aplicação PON são descritas nas subseções seguintes.

4.1.1 Configurações iniciais de uma aplicação PON

A composição de aplicações PON se dá por meio da criação e encadeamento de entidades PON, de modo que tais entidades formem um fluxo coeso e bem distribuído de notificações pontuais. Essencialmente, esse fluxo dita a maneira com que o programa se comporta e responde a eventos ocorridos no mesmo.

Ademais, a criação de entidades PON é dada pelo uso de uma fábrica de entidades, a qual possui como responsabilidade principal instanciar tais entidades adaptadas a uma estrutura de dados específica. A forma com que essas entidades são definidas e suas possíveis estruturas de dados são detalhadas nas subseções 3.1.2 e 3.1.3.

O Algoritmo 20 demonstra a inicialização dos componentes iniciais de uma aplicação PON, em especial, a classe *SingletonFactory*, responsável pela instanciação de entidades PON.

```
1 void SimpleFlightSimulator::initStartApplicationComponents() {
2   SingletonFactory::changeStructure(SingletonFactory::NOP_VECTOR);
3   SingletonLog::changeStream(SingletonLog::CONSOLE);
4   this->startApplication();
5 }
```

Algoritmo 20 – Inicialização dos componentes iniciais de uma aplicação PON

Conforme ilustra o Algoritmo 20, o método *initStartApplicationComponents* pode ser utilizado para a inicialização da fábrica concreta de entidades (linha 2). As possíveis fábricas a serem utilizadas na criação de entidades PON, até o momento da escrita desse trabalho, são *NOP_LIST*, *NOP_VECTOR*, *NOP_HASH* e *STL_LIST*. Ademais, esse método pode ser utilizado na inicialização do gerador de *logs* (linha 3). É importante observar, que os mesmos métodos *changeStructure* e *changeStream* podem ser utilizados em outras partes do código, caso o desenvolvedor ache pertinente e necessário. Por fim, conforme linha 4, o desenvolvedor deveria chamar o método *startApplication* que é responsável por chamar ordenadamente os métodos de criação de uma aplicação PON. Tal método é implementado na classe *NOPApplication*, conforme apresenta o Algoritmo 21.

```
1  void NOPApplication::startApplication() {
2     initFactBase();
3     initSharedEntities();
4     initRules();
5     codeApplication();
6  }
```

Algoritmo 21 – Implementação do método *startApplication*

Nota-se que a essência da classe *NOPApplication* é bastante simples, podendo ser inclusive ignorada na criação de uma aplicação PON. Entretanto, a extensão dessa classe é aconselhável, uma vez que segue bons princípios, ao dividir o processo de inicialização, configuração e construção de uma aplicação PON em métodos coesos e intuitivos. As próximas subseções descrevem os procedimentos para os demais passos utilizados na criação de uma aplicação PON, mencionando a implementação dos demais métodos herdados da classe *NOPApplication*.

4.1.2 Composição de *FBEs*

Conforme apresentado na Subseção 3.3.1, a estrutura da classe *FBE* sofreu algumas alterações no *Framework* PON otimizado, tais como estender a classe base *NOPEntity* e a possibilidade de vincular um *FBE* com um *FBE* pai, chamado de *masterFBE*. Tais funcionalidades proporcionam meios para que o desenvolvedor acompanhe o fluxo de execução de uma aplicação PON, proporcionando basicamente facilidades para depuração de código, tida como uma das dificuldades no âmbito da concepção de aplicações PON.

A composição de entidades *FBEs* no atual *Framework* PON se dá por meio da criação e instanciação de classes. Inicialmente, o desenvolvedor deve criar as classes que representam os modelos para criação de entidades *FBE* pertinentes à sua aplicação. Tais classes podem conter *Attributes*, *Methods*, referências para outras instâncias de *FBEs* e inclusive *Rules*. Neste sentido, o diagrama de classes ilustrado na Figura 68, apresenta uma parte da estrutura de classes do caso de estudo simulador de voo, sob o viés de composição de *FBEs*.

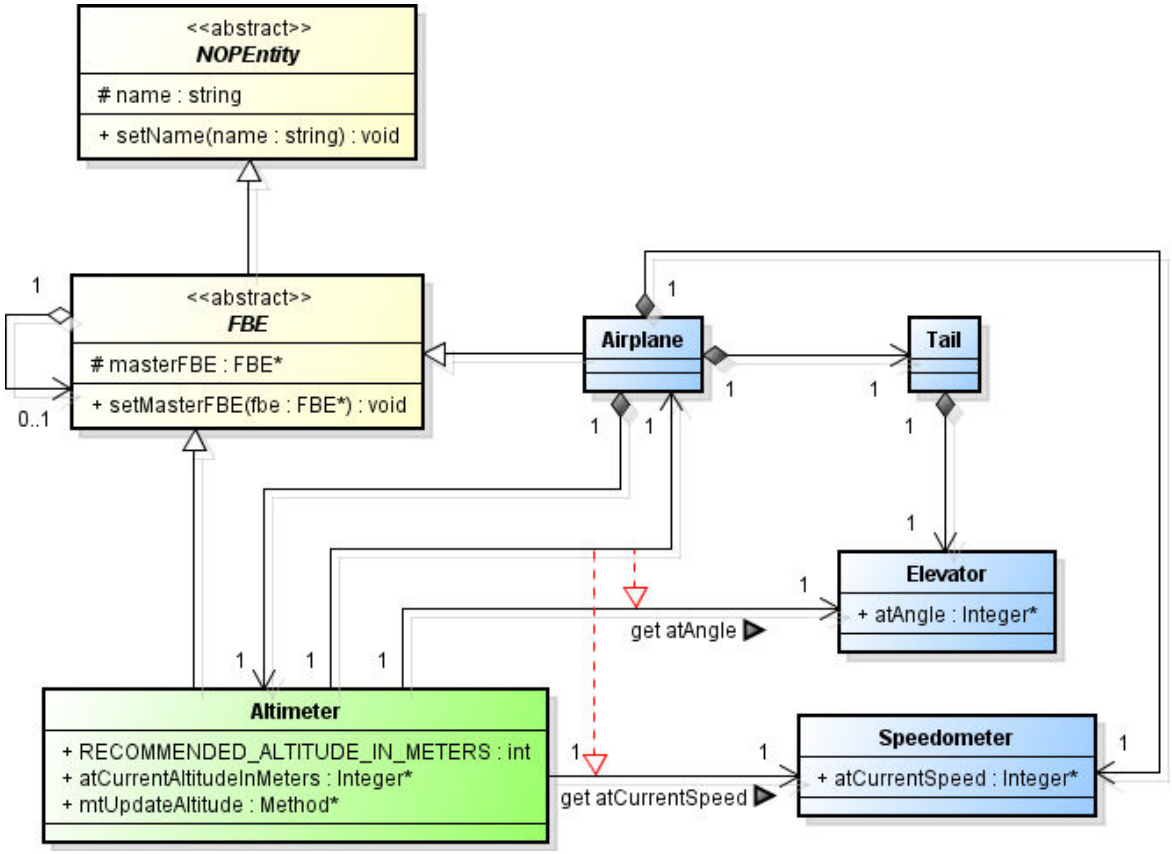


Figura 68 – Diagrama de classes do simulador de voo - composição de *FBEs*

Conforme ilustra a Figura 68, algumas classes do caso de uso simulador de voo são apresentadas. Tais classes representam os modelos dos *FBEs* a serem criados e utilizados no âmbito de concepção de uma aplicação PON.

De acordo com a sequência de criação de aplicações, o próximo método herdado da classe *NOPApplication* a ser implementado é o método *initFactBase*. Sendo assim, o Algoritmo 22 apresenta a criação de entidades *FBE* para o caso de estudo simulador de voo.

```
1  void SimpleFlightSimulator::initFactBase() {
2
3     curitibaToManaus = new Route(1000000);
4     curitibaToManaus->setName("From Curitiba -> To Manaus");
5
6     boeing737 = new Airplane("Boeing 737", 240000);
7
8     john = new Aviator("John", new PassiveBehavior());
9
10    flight1A = new Flight(curitibaToManaus, boeing737);
11    flight1A->setName("Flight 1A");
12
13    john->assignFlight(flight1A);
14
15 }
```

Algoritmo 22 – Exemplo de criação de entidades *FBEs*

Conforme ilustra o código do Algoritmo 22, na linha 3 é criado uma instância de um *FBE* do tipo *Route*. Posteriormente na linha 4, tal instância recebe um nome que será utilizado para sua identificação no gerador de *log* de execução desse programa. A linha 6, por sua vez, cria uma instância de um *FBE Airplane*, cuja nomenclatura e demais parâmetros são definidos no mesmo construtor. A linha 8, particularmente, cria uma instância de um *FBE Aviator*, a qual além de sua nomenclatura, recebe uma instância de outro *FBE* (*i.e. PassiveBehavior*).

Ademais, as linhas 10 e 11 definem uma instância de um *FBE* do tipo *Flight*, cuja referência é passada por parâmetro para um método tradicional do POO, criado para definir dependências entre instâncias de *FBEs.* É possível observar que as classes que definem os modelos dos *FBEs* não precisam seguir regras rigorosas quanto suas estruturas, tornando a programação dessas uma tarefa flexível.

Todavia, a criação das classes que modelam *FBEs* poderiam seguir alguns padrões de implementação que proporcionam maior legibilidade para o código, bem como as demais vantagens advindas de uma programação precisa. Neste sentido a subseção seguinte apresenta os padrões de implementação adequados para a composição de *FBEs*.

4.1.3 Padrões de implementação específicos para o PON

Basicamente, os padrões de implementação representam a utilização de nomenclatura adequada para com os elementos de *software*, correto uso de comentários, formatação e organização do código como um todo. A Seção 2.7 apresentou tais padrões aplicados no âmbito de aplicações orientadas a objeto, desenvolvidas com a linguagem de programação C++. Todavia, esses são perfeitamente aplicáveis em aplicações PON, uma vez que o atual *Framework* PON se apresenta materializado nessa linguagem.

Outrossim, conforme apresentado na Seção 2.7.1, a nomenclatura dos elementos de *software* são de extrema importância para o aumento da legibilidade do código como um todo. Os elementos de uma aplicação PON necessitariam ainda mais de cuidados com a nomenclatura, uma vez que representam pequenas entidades distintas e independentes, que necessitam ser ‘encaixadas’ adequadamente para que formem o fluxo de execução desse paradigma. Neste

âmbito, recomenda-se a adoção de abreviaturas dos tipos de entidades PON, prefixadas no nome das instâncias de tais entidades, com o objetivo de facilitar a distinção entre essas, facilitando ademais suas posteriores 'junções'. Sendo assim, a Tabela 3 apresenta os prefixos recomendados para a nomenclatura de entidades PON.

Tabela 3 – Prefixos para a nomenclatura de entidades PON

Entidade PON	Prefixo	Exemplo
Attribute	*at*	*atCurrentAltitudeInMeters*
Premise	*pr*	*prAirplaneIsTurnedOn*
Condition	*cd*	*cdCheckGameOver*
Rule	*rl*	*rlValidateCreditAndApproveSale*
Action	*ac*	*acApproveSale*
Instigation	*in*	*inClearMaze*
Method	*mt*	*mtUpdateAltitude*

Em relação à formatação e organização do código, os padrões adotados na concepção dos casos de estudo aqui expostos seguem, em geral, o padrão de implementação ilustrado no Algoritmo 23. Nesse são apresentados o código do cabeçalho do *FBE Altimeter* (*i.e.* um dos componentes do *FBE Airplane*) do caso de estudo simulador de voo.

```
1  class Altimeter : public FBE {
2
3     public:
4
5        const static int RECOMMENDED_ALTITUDE_IN_METERS = 10000;
6
7     public:
8
9        Altimeter(Airplane *airplane);
10       virtual ~Altimeter();
11
12    public:
13
14       Integer *atCurrentAltitudeInMeters;
15
16       Method *mtUpdateAltitude;
17
18 };
```

Algoritmo 23 – Implementação da estrutura da classe *Altimeter*

Recomenda-se adotar o padrão de implementação ilustrado no Algoritmo 23 para a definição do 'esqueleto' de um *FBE* em seu arquivo de cabeçalho. A implementação deveria ser feita de maneira que a(s) constante(s) de um *FBE* fiquem

separada(s) do(s) construtor(es) e destrutor, bem como de suas entidades *Attributes* e *Methods*. Todas as entidades que compõem um *FBE* e que geralmente são acessadas externamente por outros elementos, por questões de praticidade e desempenho, são definidas como públicas, evitando assim chamadas desnecessárias a métodos encapsuladores.

A implementação concreta de tal *FBE* consiste na definição de suas particularidades, tais como *name, masterFBE*, *Attributes* e *Methods*, geralmente definidos no construtor dessa classe. Ademais, a estrutura de um *FBE* pode conter a implementação de eventuais métodos OO, quando tal recurso se fizer necessário. Particularmente, este trabalho visa uma programação PON mais purista e, portanto, proporciona aos desenvolvedores um conjunto de padrões para atingir um código menos dependente de implementações multiparadigmas.

O Algoritmo 24 apresenta as particularidades de implementação do *FBE Altimeter,* fazendo uso dos novos recursos descritos anteriormente.

```
 1 Altimeter::Altimeter(Airplane *airplane) {
 2
 3    this->setName("Altimeter");
 4
 5    this->setMasterFBE(airplane);
 6
 7    INTEGER(this, this->atCurrentAltitudeInMeters, 0);
 8
 9    METHOD_OPERATION(
10       this, // FBE em questão
11       this->mtUpdateAltitude,
12       this->atCurrentAltitudeInMeters,
13       (new Addition(
14          this->atCurrentAltitudeInMeters,
15          new Multiplication(
16             new Sine(airplane->tail->elevator->atAngle),
17             airplane->speedometer->atCurrentSpeed))),
18        Attribute::STANDARD);
19
20 }
```

Algoritmo 24 – Particularidades de implementação do *FBE Altimeter*

Conforme apresenta o código do Algoritmo 24, a linha 3 define a nomenclatura para este *FBE*. Na linha 5, por sua vez, o método *setMasterFBE* define a dependência do *FBE Altimeter* com o *FBE Airplane*. Essa dependência é necessária para a implementação da entidade *mtUpdateAltitude*, uma vez que essa utiliza como base para o cálculo de altitude, as entidades *Attribute* de outros *FBEs* dependentes de *Airplane*.

Ainda, o *FBE Altimeter* possui um *Attribute* do tipo *Integer*. A linha 7 do Algoritmo 24 demonstra um exemplo de utilização do pseudônimo na criação do *Attribute atCurrentAltitudeInMeters*, inicializado com o valor 0.

Ademais, as linhas 9 a 18 demonstram um exemplo de utilização do pseudônimo na criação do *Method mtUpdateAltitude*. Detalhadamente os parâmetros definidos para esse são: o respectivo *FBE* (*i.e. Altimeter*) no qual tal *Method* é criado (linha 10); o próprio ponteiro para o *Method* (linha 11); o respectivo *Attribute* que receberá o resultado da operação (linha 12); a respectiva operação aritmética composta (linhas 13 a 17); e, por fim, uma *flag* que determina o comportamento padrão para o *Attribute*.

4.1.4 Particularidades de entidades *Attribute*

Em geral, *Attributes* proporcionam grande impacto na concepção de aplicações PON, uma vez que representam o ponto de partida para a realização do calculo lógico-causal do PON. Neste sentido, alguns cuidados com a criação deveriam ser tomados para atingir um nível de programação eficiente e correta. Para isso, a Subseção 4.1.4.1 apresenta os comportamentos distintos de um *Attribute* e formas de controlar tais comportamentos. A Subseção 4.1.4.2, por sua vez, apresenta a utilização de *Attributes* ‘impertinentes’ em um caso de estudo real, bem como apresenta os resultados de um comparativo de desempenho realizado em tal caso de estudo.

4.1.4.1 Definição do comportamento reativo de um *Attribute*

De maneira geral, um *Attribute* pode apresentar três comportamentos distintos ao ter seu estado alterado. O comportamento padrão, de acordo com a teoria, é notificar as *Premises* interessadas somente quando o estado de tal *Attribute* tiver sofrido alterações. Todavia, existem situações que demandam que uma *Rule* seja reavaliada e executada novamente, mesmo quando os estados de suas entidades colaboradoras não tenham sofrido alterações. Ainda, existem casos em

que as alterações no estado de um *Attribute* não deveriam gerar notificações em nenhuma das situações.

As alterações no estado de um *Attribute* podem ocorrer em qualquer parte do código através de implementações diretas baseadas no POO ou mais recentemente na utilização do recurso de *Methods* PON. Neste âmbito, para definir o modo com que um *Attribute* vai reagir perante a alterações em seu estado, o Algoritmo 25 apresenta exemplos de código utilizando *flags* de definição de comportamento.

```
1  atStatus->setValue(true, Attribute::STANDARD);
2
3  atTemperature->setValue(20.3, Attribute::RENOTIFY);
4
5  atAge->setValue(25, Attribute::NO_NOTIFY);
```

Algoritmo 25 – *Flags* para a definição do comportamento dos *Attributes*

Conforme ilustrado no código do Algoritmo 25, a linha 1 representa a atribuição padrão de um valor para um dado *Attribute*, o que resultaria no disparo do fluxo de notificações caso o estado desse apresente um valor diferente do anterior. A linha 3, por sua vez, representa a atribuição baseada em renotificações, disparando o fluxo de notificações sempre independente do estado anterior do *Attribute* em questão. A linha 5, particularmente, apresenta a atribuição para casos em que o disparo do fluxo de notificações seja indesejado. Outrossim, caso a *tag* seja omitida, o comportamento padrão será adotado.

4.1.4.2 Utilização de *Attributes* ‘impertinentes’

De maneira geral, uma situação em que *Attributes* ‘impertinentes’ apareceriam seria na *Rule* responsável por iniciar o processo de aterrissagem da aeronave, considerando o caso de estudo simulador de voo (Subseção 2.4.3). Essa *Rule* em questão teria em seu escopo as seguintes *Premises*: (a) pedido de autorização para aterrissagem aprovado? (b) altitude atual da aeronave superior a

500 metros? e (c) velocidade da aeronave superior a 400 quilômetros por hora?[7]. Após a aprovação da *Rule* mencionada a respectiva aeronave entraria em modo de pouso. Tal cenário é ilustrado na Figura 69.

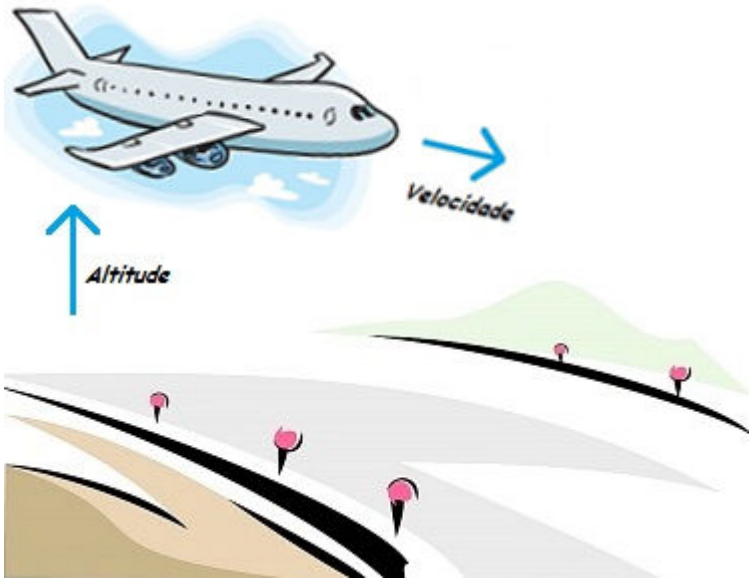


Figura 69 – Cenário de um ambiente com *Attributes* 'impertinentes'

É possível observar que o cenário descrito apresentaria a presença de *Attributes* 'impertinentes', uma vez que a altitude e velocidade de uma aeronave estariam variando de estado constantemente. Neste caso, tais *Attributes* ao serem classificados como do tipo 'impertinente' deixariam de notificar as *Premises* dessa *Rule* exemplificada, habilitando suas respectivas notificações apenas quando o estado da primeira *Premise* (*i.e.* pedido de autorização para aterrisagem aprovado) apresentar o estado verdadeiro.

De modo a exemplificar a implementação de tal funcionalidade, o código do Algoritmo 26 apresenta o uso dessa na criação das entidades descritas no cenário exemplo.

[7] Este seria um exemplo hipotético de uma aeronave em estado de voo normal (altitude e velocidade superiores as normalmente empregadas no estado de aterrissagem), a espera de uma autorização para a realização da aterrissagem.

```
1  Boolean *atLandRequestApproved =
2     SingletonFactory::getInstance()->createBoolean(
3        airplane, false, Attribute::STANDARD);
4
5  Boolean *atCurrentSpeed =
6     SingletonFactory::getInstance()->createDouble(
7        airplane, , 0.0, Attribute::IMPERTINENT);
8
9  Boolean *atCurrentAltitude =
10    SingletonFactory::getInstance()->createDouble(
11       airplane, , 0.0, Attribute::IMPERTINENT);
```

Algoritmo 26 – Exemplo do uso de *Attributes* 'impertinentes'

O código descrito no Algoritmo 26 representa a criação das entidades 'impertinentes' descritas no cenário exemplo (aterrissagem da aeronave). O desenvolvedor possui como única responsabilidade determinar o tipo de cada entidade *Attribute* criada. Ao compor *Premises* com *Attributes* do tipo 'impertinente', o *framework* controla todo o processo de ativação e desativação das notificações automaticamente, conforme descrito anteriormente.

De maneira a analisar a eficiência e quantificar os benefícios do uso de *Attributes* 'impertinentes' no âmbito da concepção de aplicações PON, um experimento comparativo foi realizado no caso de estudo simulador de voo simples. O teste representa o cenário exemplo ilustrado, no qual uma regra adicional representa a condição para o pouso da aeronave (considerando a ordem para aterrissagem, velocidade e altitude). Assim, a Figura 70 apresenta a comparação entre a implementação tradicional e a implementação que faz uso dos *Attributes* 'impertinentes'.

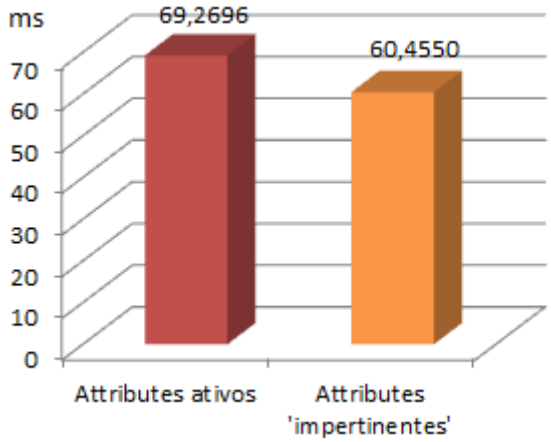


Figura 70 – Comparativo cenário exemplo - *Attributes* ativos x 'impertinentes'

A Figura 70 apresenta a eficiência proporcionada pela utilização de *Attributes* ‘impertinentes’ no cenário exemplo, a qual representa aproximadamente um ganho de desempenho de cerca de 14,5% em relação à implementação padrão (*Attributes* ativos).

De maneira detalhada, nos experimentos ocorreram cerca de 3000 alterações nos valores dos dois *Attributes* (velocidade e altitude). Tais *Attributes* só se apresentaram pertinentes à *Rule* em questão apenas nas 3 últimas alterações de valores dos mesmos. Sendo assim, na grande maioria das vezes realizaram (ou realizariam, no caso da utilização de *Attributes* ‘impertinentes’) notificações desnecessárias.

É importante ressaltar que os tempos medidos representam uma execução completa da aplicação, incluindo a ativação/execução de todas as *Rules* dessa. Neste sentido, caso o *Attribute* impertinente fosse considerado isoladamente para com a *Rule* marcada como passível de impertinência, tal *Premise-Conditon-Rule* não receberia nenhuma notificação, enquanto sem esse recurso receberia *x* notificações (onde *x* representaria o número de vezes que o estado do *Attribute* variou). Nesse âmbito, a diferença de desempenho entre as implementações é de 0 (zero) para *y* (onde *y* representaria o tempo de processamento consumido para a realização de *x* notificações).

4.1.5 Particularidades de entidades *Method*

As versões anteriores do *Framework* PON apenas possibilitavam a criação de entidades *Method* vinculadas a métodos tradicionais do POO. De certo modo, tal prática possibilita a inserção de qualquer tipo de estrutura desse paradigma, o que implicaria na possibilidade de inserção de redundâncias e demais problemas, conforme detalhado na Seção 2.1. Além disso, mesmo simples instruções como a alteração do valor de um *Attribute*, ou até mesmo a adição de um valor a um *Attribute* existente, se apresentavam como tarefas assaz complexas.

Neste âmbito, a nova estrutura do *framework* apresenta várias formas distintas para a composição de entidades *Method*. Assim, atividades simples como atribuições ou operações aritméticas apresentam interfaces mais amigáveis para suas concepções.

De maneira a demonstrar a complexidade em se criar uma entidade *Method* nas versões anteriores do *framework,* o Algoritmo 27 ilustra um exemplo dessa implementação no caso de estudo simulador de voo.

```
1  // FBE parameters
2  Parameters *parameters = new Parameters();
3  parameters.addElement(airplane->tail->leftElevator->atAngle);
4  parameters.addElement(airplane->speedometer->atCurrentSpeed);
5
6  // Rule PON
7  rlAccelerateAirplane->addInstigation(
8              new Instigation(mtUpdateAltitudeOOP, parameters));
9
10 // Método POO
11 void Altimeter::updateAltitude(Parameters *parameters){
12
13    Integer *atAngle = (Integer*) parameters->getElement(0);
14    Integer *atSpeed = (Integer*) parameters->getElement(1);
15
16    int altitude = (atCurrentAltitudeInMeters->getValue() +
17                   (sin(angle->getValue()) * speed->getValue()));
18
19    this->atCurrentAltitudeInMeters->setValue(altitude);
20
21 }
22
23 // MethodPointer PON
24 Method *mtUpdateAltitudeOOP =
25             new MethodPointer<Altimeter>(this,
26                                 &Altimeter::updateAltitude);
```

Algoritmo 27 – Exemplo de criação e vinculação de métodos POO

Conforme apresenta o Algoritmo 27, as linhas 2 a 4 exemplificam a maneira com que os parâmetros de um método são armazenados em uma estrutura de dados para posterior uso. As linhas 7 e 8, por sua vez, exemplificam a maneira com que um *MethodPointer* é adicionado a uma *Rule*, relacionando-o com seus respectivos parâmetros. As linhas 11 a 21, particularmente, exemplificam a criação de um método do POO, com o procedimento para a realização de um cálculo aritmético simples. Por fim, as linhas 24 a 26 exemplificam a vinculação de um *Method* PON com um método do POO.

É possível observar que a implementação de operações aritméticas no PON, seguindo o modelo antigo, apresenta complexidade excessiva em sua concepção, bem como os demais problemas relatados.

De modo a exemplificar a utilização do novo *Method* PON na estrutura do *Framework* PON, o Algoritmo 28 ilustra um exemplo de utilização de tal funcionalidade na implementação da mesma operação apresentada no Algoritmo 27.

```
1  METHOD_OPERATION(this,
2     mtUpdateAltitudeNOP,
3     atCurrentAltitudeInMeters,
4     (new Addition(
5         atCurrentAltitudeInMeters,
6         new Multiplication(
7            new Sin(airplane->tail->leftElevator->atAngle),
8            airplane->speedometer->atCurrentSpeed))),
9     Attribute::RENOTIFY);
```

Algoritmo 28 – Exemplo de utilização da funcionalidade *Method Operations*

Conforme apresenta o Algoritmo 28, a composição de *Methods* PON teve sua essência simplificada, ao passo que elimina a possibilidade de adição de redundâncias estruturais e temporais na estrutura das aplicações PON.

Neste âmbito, o uso de tal funcionalidade proporciona benefícios como facilidade na composição de aplicações, clareza no código através de maior legibilidade, que por sua vez deixa o código mais manutenível e reaproveitável. Além disso, o código em questão se torna mais puro ao evitar o uso de programação multiparadigmas, trazendo benefícios outros como facilidades na distribuição do código, bem como na composição de programas através da ferramenta *Wizard* PON.

Em contra partida, a adição de funcionalidades facilitadoras tende a impactar no desempenho de execução de uma aplicação. Neste caso, cabe ao desenvolvedor optar pela melhor abordagem que se adéque à suas necessidades.

4.1.6 Composição de *Rules*

A composição de *Rules* é a etapa que define o fluxo de execução de uma aplicação PON. Apesar de todas as entidades PON contribuírem ativamente para gerar o fluxo de execução de tal aplicação, são as *Rules* que definem os relacionamentos entre essas. Neste sentido, a composição de *Rules* é uma tarefa que exige certos cuidados, pois a aprovação de uma *Rule* pode impactar na não execução de outra *Rule*, mesmo que aprovada *a priori*. Esse problema tende a se agravar com um número grande de *Rules*, tornando a programação nesse paradigma dificultosa.

4.1.6.1 Utilização de *Rules* dependentes

A utilização do recurso de dependência entre *Rules* se mostra bastante útil em casos onde duas ou mais *Premises* precisam ser compartilhadas por um número considerável de outras *Rules*. Neste caso, tais *Premises* fariam parte de uma única *Rule*, a qual faria o papel de *Rule* mestre. As demais *Rules* fariam um vínculo com essa *Rule*, de tal forma que dependeriam de sua aprovação para só então executarem. Essa dependência traria benefícios em questões de desempenho e facilidades na composição de aplicações.

No âmbito de eficiência na execução, tais *Rules* atuariam na redução de notificações geradas pelas *Premises* em questão, que direcionariam suas notificações apenas à *Rule* mestre. Em relação à facilidade de composição de aplicações, tal abordagem simplificaria a essência de todas as demais *Rules*, tornando o código mais legível e consequentemente mais manutenível.

A fim de exemplificar a aplicação dessa funcionalidade, o caso de estudo Simulador de voo apresenta um cenário no qual o aviador depende de uma *Rule* mestre para executar todas as suas ações (*i.e.* maior parte das *Rules* dessa aplicação). O Algoritmo 29 apresenta a essência de tal *Rule*.

```
1 RULE(rlAviatorCanOperate, scheduler, Condition::CONJUNCTION);
2 rlAviatorCanOperate->addPremise(prAirplaneIsTurnedOn);
3 rlAviatorCanOperate->addPremise(prAirplaneHasNotReachedItsDestine);
```

Algoritmo 29 – Exemplo de *Rule* mestre

Conforme ilustrado no Algoritmo 29, a *Rule* em questão verifica se o avião está ligado e se ele ainda não chegou ao seu destino. É possível observar que a *Rule* em questão não apresenta instigações. Isso ocorre justamente porque essa *Rule* apresenta como única responsabilidade a de servir de *Rule* mestre para as demais *Rules* da aplicação. Ainda, tal *Rule* poderia ter instigações em sua essência caso essas se fizerem necessárias.

Outrossim, essa *Rule* apresenta outra característica importante no âmbito de concepção de aplicações PON. Em conjunto com o recurso de renotificações, tal *Rule* atua como uma *Rule* iteradora. A cada iteração realizada por essa, o valor da distância percorrida pelo avião é atualizada e, por manter o fluxo baseado em

renotificações, mesmo que o estado dessa permaneça aprovado, a *Rule* em questão é reaprovada e executada até que a condição de parada seja alcançado. Nesse caso, tanto o desligar do avião, quanto o alcançar do destino, representariam tal condição de parada.

Ainda, as demais *Rules* que compõem a pilotagem do avião são dependentes da *Rule* iteradora apresentada, realizando todas as devidas ações a cada iteração gerada, até que o voo seja completado. De maneira a exemplificar um exemplo de *Rule* dependente, a Figura 71 apresenta a essência da *Rule* que controla a aceleração do avião.

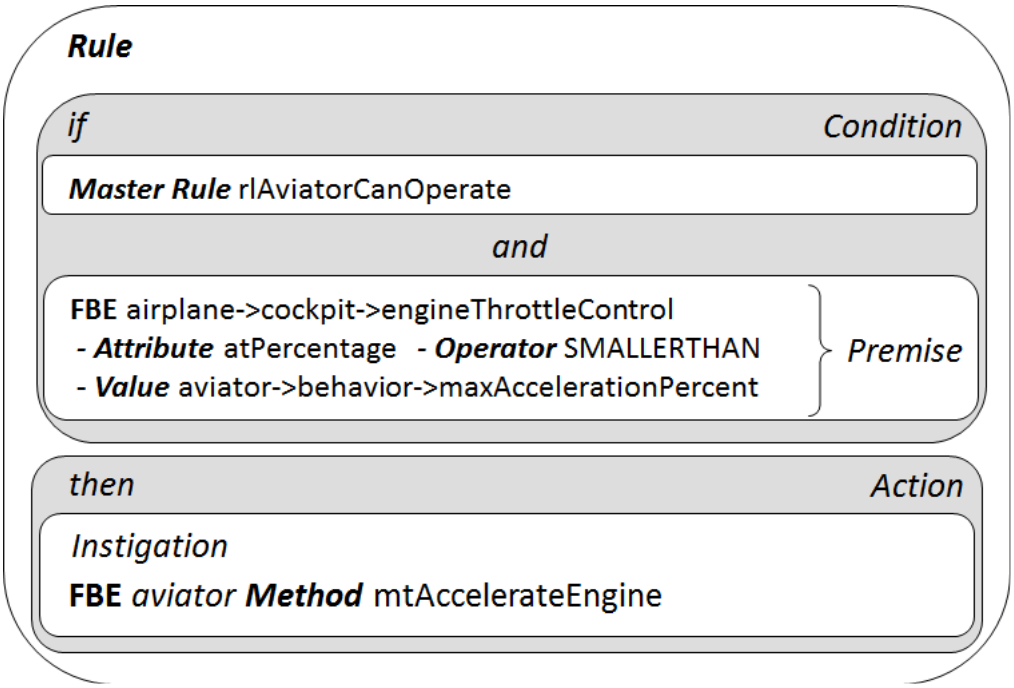


Figura 71 – Exemplo de *Rule* dependente

Conforme ilustra a *Rule* representada na Figura 71, a aceleração do avião ocorre quando o mesmo não se encontra totalmente acelerado. Tal aceleração é definida pelo comportamento do piloto (*e.g.* passivo ou agressivo) atribuído ao voo em questão. Ainda, tal *Rule* apresenta uma dependência para com a *Rule* iteradora *rlAviatorCanOperate*, sendo executada apenas quando a *Rule* mestre tiver sido executada. De modo a apresentar a codificação de uma *Rule* dependente, o Algoritmo 30 expõe a essência da *Rule* ilustrada na Figura 71.

```
1 RULE(rlAviatorAccelerateEngine, scheduler, Condition::CONJUNCTION);
2 rlAviatorAccelerateEngine->addMasterRule(rlAviatorCanOperate);
3 rlAviatorAccelerateEngine->addPremise(
4    airplane->cockpit->engineThrottleControl->atPercentage,
5    aviator->behavior->maxAccelerationPercent,
6    Premise::SMALLERTHAN, Attribute::STANDARD);
7 rlAviatorAccelerateEngine->addMethod(aviator->mtAccelerateEngine);
```

Algoritmo 30 – Exemplo de código de uma *Rule* dependente

Conforme apresentado no Algoritmo 30, a linha 2 apresenta o método que vincula uma *Rule* dependente à uma *Rule* mestre. As demais entidades de uma *Rule* seguem o procedimento normal de composição. Outrossim, a dependência entre *Rules* principalmente simplifica a visualização do código e entendimento do mesmo como um todo.

4.1.7 Compartilhamento de entidades PON

Sucintamente, a boa prática de compartilhar entidades PON apresenta grande importância, tanto em questões de facilidade de desenvolvimento, quanto em questões de desempenho. O correto uso do compartilhamento de entidades PON eliminaria a criação de entidades redundantes, bem como possíveis notificações desnecessárias para essas.

De modo a exemplificar o uso de tal boa prática, considera-se o seguinte cenário no âmbito do simulador de jogo *Pacman*. Conforme descrito na Subseção 2.4.1, a movimentação dos personagens no labirinto ocorre apenas quando esses se encontram em determinadas posições do labirinto (*i.e.* esquinas). Conjuntamente com a localização dos personagens no labirinto, os elementos percebidos por esses influenciam em suas tomadas de decisão (*i.e.* escolha de direção). Tanto as *Premises* que verificam suas posições atuais, quanto as *Premises* que verificam os elementos percebidos por esses são entidades presentes em mais de uma *Rule*, portanto passíveis de compartilhamento.

De modo a ilustrar um exemplo desse cenário, o Algoritmo 31 apresenta um trecho de código do simulador de jogo *Pacman*. Pertinentemente, o exemplo ilustrado representa o código descrito na Figura 20, apresentado anteriormente na descrição do caso de estudo desse simulador (Subseção 2.4.1).

```
 1 PREMISE(prPacmanAtCornerDownRight,
 2    pacman->atCurrentCorner, Corner::DownRight, Premise::EQUAL);
 3
 4 . . .
 5
 6 PREMISE(prPacmanSpottedGhostAtRight,
 7    pacman->atElementSpotted, Element::GhostAtRight, Premise::EQUAL);
 8
 9 PREMISE(prPacmanSpottedDotAtRight,
10    pacman->atElementSpotted, Element::DotAtRight, Premise::EQUAL);
11
12 . . .
13
14 METHOD(mtChangeDirectionToRight,
15    pacman->atDirection, Character::RIGHT, Attribute::STANDARD);
16
17 METHOD(mtChangeDirectionDownward,
18    pacman->atDirection, Character::DOWNWARD, Attribute::STANDARD);
19
20 . . .
21
22 RULE(rlRunAwayFromGhost, scheduler, Condition::CONJUNCTION);
23 rlRunAwayFromGhost->addPremise(prPacmanAtCornerDownRight);
24 rlRunAwayFromGhost->addPremise(prPacmanSpottedGhostAtRight);
25 rlRunAwayFromGhost->addInstigation(mtChangeDirectionDownward);
26
27 RULE(rlAbsorbDot, scheduler, Condition::CONJUNCTION);
28 rlAbsorbDot->addPremise(prPacmanAtCornerDownRight);
29 rlAbsorbDot->addPremise(prPacmanSpottedDotAtRight);
30 rlAbsorbDot->addInstigation(mtChangeDirectionToRight);
```

Algoritmo 31 – Exemplo de compartilhamento de entidades PON

Conforme ilustrado no Algoritmo 31, a primeira *Premise* (linhas 1 e 2) é compartilhada com duas *Rules* distintas (nas linhas 23 e 28). É importante ressaltar que apesar de não estar estritamente detalhado no código exemplo, o simulador de jogo *Pacman* apresenta uma grande quantidade de *Rules*, as quais possuem muitas entidades passíveis de compartilhamento. A título de exemplo, os personagens só podem optar pela movimentação entre quatro direções distintas. Neste sentido, seria necessário criar apenas quatro entidades *Methods* e compartilhá-las com as *Rules* pertinentes.

Outrossim, o artigo [SIMÃO *et al.,* 2012c], afixado no Apêndice A, apresenta comparativos quantitativos entre duas implementações do simulador de jogo *Pacman*. Apesar de não estar explícito o impacto do compartilhamento de *Premises* nos resultados apresentados nesse artigo, devido a outras questões como melhorias no *framework* e mudanças no ambiente de testes, foi possível observar que o modo como a aplicação foi reimplementada teve grande impacto no desempenho.

Em suma, as contribuições em geral para com os novos experimentos levaram a implementação do simulador, desenvolvido sob os princípios do PON, a obter resultados bastante favoráveis, se aproximando do desempenho obtido pela mesma implementação sob os princípios do POO. Em suma, Valença (2012) demonstrou que a implementação do novo *Framework* PON apresentou ganhos de desempenho de cerca de 2 vezes em média. No artigo [SIMÃO *et al.,* 2012c], em especial, os ganhos para com a versão anterior se mostraram ainda mais favoráveis, apresentando um desempenho cerca de 5 vezes superior.

4.1.8 Controle do fluxo de execução no PON

De maneira geral, outra dificuldade de implementação advinda do fluxo de execução não tradicional ditado pelo processo de notificações entre entidades PON é manter o controle do fluxo de execução de uma aplicação PON. Isso ocorre justamente pela possível falta de sequencialidade na mesma, que pode inclusive levar a uma execução indeterminística (*i.e.* dada uma mesma entrada, apresentar execuções com diferentes saídas).

De maneira a exemplificar o modo não tradicional de execução das aplicações PON, considere o seguinte cenário apresentado no simulador de jogo *Pacman:* ambos os personagens, *Ghost* e *Pacman* caminhando na mesma direção, gerando uma colisão no centro do labirinto. Nesse cenário, ocorrerão duas colisões, uma entre o *Pacman* e a pastilha e outra entre o *Ghost* e o *Pacman*. Tal cenário é ilustrado na Figura 72.

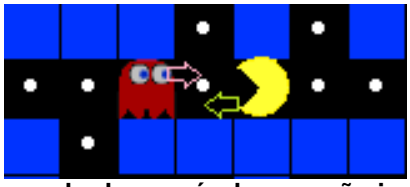


Figura 72 – Exemplo de possível execução indeterminística

Os algoritmos Algoritmo 32 e Algoritmo 33 apresentam respectivamente o mesmo trecho de código do simulador de jogo *Pacman* implementado tanto no POO quanto no PON.

```
 1  void gameLoop() {
 2     checkCollisions();
 3     handleCollisions();
 4  }
 5
 6  void checkCollisions() {
 7
 8     . . . // Additional collision checking.
 9
10     for (int i = 0; i < ghosts.size; i++)
11        if (pacman.position == ghosts[i].position)
12           collidedWithGhost = true;
13
14     for (int i = 0; i < dots.size; i++)
15        if (pacman.position == dots[i].position)
16             if (dots[i].type == NORMAL)
17              collidedWithNormalDot = true;
18
19        . . . // Additional collision checking.
20
21  }
22
23  void handleCollisions() {
24
25     . . . // Additional collision handling.
26
27     if (collidedWithNormalDot) {
28        game->score->value += 10;
29     }
30
31     . . . // Additional collision handling.
32
33     if (collidedWithGhost) {
34        game->score->numLives -= 1;
35        repositionCharacters();
36     }
37
38  }
```

Algoritmo 32 – Exemplo de código POO no simulador do jogo *Pacman*

Considerando o exemplo do Algoritmo 32, no qual as instruções são executadas sequencialmente, o método *gameLoop* (linha 1), tido como método principal no âmbito dessa aplicação, direciona o fluxo de execução para o método *checkCollisions* (linha 6) e por conseguinte para o método *handleCollisions* (linha 23). Outrossim, em tais métodos todas as instruções são realizadas sequencialmente, não sofrendo impacto direto no fluxo de execução.

Neste ínterim, tanto a colisão com a pastilha, quanto a colisão com o *Ghost* ocorreriam e seriam devidamente tratadas pelo código desenvolvido no POO. Em contra partida, o mesmo exemplo desenvolvido sob os princípios do PON poderia apresentar problemas de execução, dependendo da maneira com que as *Rules*

forem concebidas e ordenadas (no momento de suas criações). Para maiores detalhes o Algoritmo 33 ilustra a presença de tal problema.

```
1  RulePlus* rlCheckCollisionBetweenPacmanAndGhost;
2  rlCheckCollisionBetweenPacmanAndGhost->addPremise(
3     pacman->atPosition, ghost->atPosition, Premise::EQUAL);
4  rlCheckCollisionBetweenPacmanAndGhost->addMethod(
5     game->atCollisionBetweenPacmanAndGhost, true);
6
7  RulePlus* rlCheckCollisionBetweenPacmanAndDot;
8  rlCheckCollisionBetweenPacmanAndDot->addPremise(
9     pacman->atPosition, dot->atPosition, Premise::EQUAL);
10 rlCheckCollisionBetweenPacmanAndDot->addMethod(
11    game->atCollisionBetweenPacmanAndDot, true);
12
13 . . .
14
15 RulePlus* rlHandleCollisionBetweenPacmanAndDot;
16 rlHandleCollisionBetweenPacmanAndDot->addPremise(
17    game->atCollisionBetweenPacmanAndDot, true, Premise::EQUAL);
18 rlHandleCollisionBetweenPacmanAndDot->addMethod(
19    game->score->atValue, IntegerPlus::ADDITION, 10);
20
21 RulePlus* rlHandleCollisionBetweenPacmanAndGhost;
22 rlHandleCollisionBetweenPacmanAndGhost->addPremise(
23    game->atCollisionBetweenPacmanAndGhost, true, Premise::EQUAL);
24 rlHandleCollisionBetweenPacmanAndGhost->addMethod(
25    game->score->mtDecreaseNumLives);
26 rlHandleCollisionBetweenPacmanAndGhost->addMethod(
27    game->mtRepositionCharacters);
```

Algoritmo 33 – Exemplo de código PON no simulador do jogo Pacman

É importante ressaltar que a ordem com que as estruturas causais foram criadas em ambas as implementações são definidas igualmente. Entretanto, a execução do código ilustrado no Algoritmo 33, considerando que nenhuma estratégia de resolução de conflitos tenha sido adotada (*i.e. NO_ONE*), irá inicialmente checar a colisão entre os personagens *Ghost* e *Pacman*, na primeira *Rule* (linhas 1 a 5), dada sua precedência na ordem de criação de *Rules*. Neste âmbito, a quarta *Rule* (linhas 21 a 27) seria aprovada no instante em que a primeira *Rule* fosse executada, dado que o *Premise* para aprovação de tal *Rule* seria aprovado no executar da primeira *Rule*. Esse cenário é ilustrado na Figura 73.

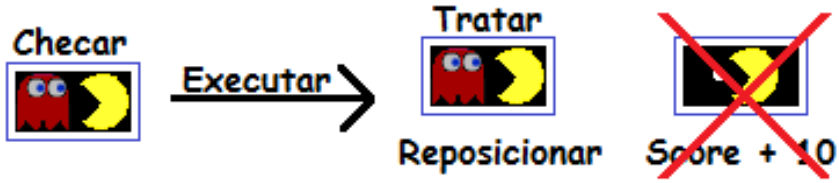


Figura 73 – Fluxo de execução sem uma estratégia de escalonamento

De maneira geral, conforme ilustrado na Figura 73, a falta de um escalonador, bem como a ordem com que as Rules foram criadas, levariam a uma mudança abrupta no fluxo de execução dessa aplicação. Isso ocorreria porque ao finalizar a execução da quarta *Rule*, o personagem *Pacman* teria sua posição alterada, devido à execução do *Method mtRepositionCharacters*, o qual reposicionaria os personagens em suas posições iniciais. Isso impactaria na desaprovação da *Premise* que considerava a colisão entre o *Pacman* e a pastilha (aprovada anteriormente) e consequentemente deixaria de considerar o tratamento dessa colisão (segunda e terceira *Rules*). Isso resultaria em um fluxo de execução diferente do realizado pela implementação OO, a qual teria todas as checagens de colisões realizadas primeiramente, para só então adentrar no respectivo método de tratamento de colisões.

Outrossim, é possível observar também que a implementação do simulador no PON poderia levar a aplicação a realizar um fluxo de execução indeterminístico e consequentemente inconsistente, uma vez que o tratamento de colisões seria realizado de maneira imperfeita, considerando as normas definidas para a correta execução do simulador.

Neste âmbito, duas soluções são apresentadas para a garantia de um fluxo de execução correto e determinístico. A Subseção 4.1.8.1 considera a utilização de *Attributes* de controle de fluxo, enquanto a Subseção 4.1.8.2 considera a utilização de escalonamento de *Rules*.

4.1.8.1 *Attributes* de Controle de Fluxo

Uma alternativa para a solução do problema descrito anteriormente seria a utilização de um *Attribute* alternativo para o controle do fluxo da execução da aplicação. Essa solução se assemelharia com a implementação provida para o POO, onde o controle de fluxo fora delegado para o método *gameLoop* (linha 1) do Algoritmo 32.

De maneira geral, a solução seria criar um *Attribute* responsável por armazenar o estado atual do jogo. Neste âmbito, um *Attribute* para o *FBE Game*, denominado *atState* poderia ser criado. Tal *Attribute* poderia alternar entre os

estados *CollisionsChecking* e *CollisionsHandling*, de modo a possibilitar com que todas as *Rules* pertinentes ao estado de checagens de colisões sejam aprovadas e executadas, igualmente para as *Rules* pertinentes ao estado de tratamento de colisões. O Algoritmo 34 apresenta a solução proposta para o controle do fluxo da aplicação.

```
1  rule->addPremise(
2     game->atState, Game::COLLISIONS_CHECKING, Premise::EQUAL);
3
4  rule->addPremise(
5     game->atState, Game::COLLISIONS_HANDLING, Premise::EQUAL);
```

Algoritmo 34 – *Attributes* de Controle de Fluxo

Conforme apresenta o Algoritmo 34, as *Rules* poderiam incluir em seu escopo as *Premises* descritas nessa solução. Sendo assim, tanto as *Rules* responsáveis pela checagem e colisão com fantasmas, quanto as responsáveis pela checagem e colisão com pastilhas, seriam propriamente executadas, uma vez que o *Attribute* proposto permitiria a execução sequencial da aplicação.

4.1.8.2 Escalonamento de *Rules*

De modo geral, uma alternativa para o controle do fluxo de execução das aplicações seria a correta utilização dos escalonadores de *Rules*, os quais teriam a função de organizar a execução de um conjunto de *Rules* previamente aprovadas. Neste âmbito, a ordem com que as *Rules* são criadas e inicializadas no *Framework* PON, bem como a escolha do mecanismo de escalonamento adequado, impactaria diretamente na ordem com que as *Rules* seriam executadas.

Ainda de acordo com o Algoritmo 33 previamente apresentado, a ordem com que as *Rules* foram criadas e inicializadas impactaram em um fluxo de execução inconsistente, ignorando a colisão existente entre o personagem *Pacman* com a pastilha no centro do labirinto. Isso ocorreu justamente pela ausência de um método de escalonamento adequado aplicado a essa execução em particular.

Sucintamente, o *Framework* PON apresenta três métodos de escalonamento (*i.e. BREADTH*, *DEPTH* e *PRIORITY*), previamente apresentados na Subseção 2.2.1. Ainda considerando o exemplo do Algoritmo 33, o escalonador ideal para a

correta execução daquele caso seria o *DEPTH*, uma vez que esse consideraria a execução seguindo uma estrutura de dados do tipo pilha. A Figura 74 ilustra o fluxo de execução do cenário exemplo considerando a estratégia de escalonamento do tipo *DEPTH*.

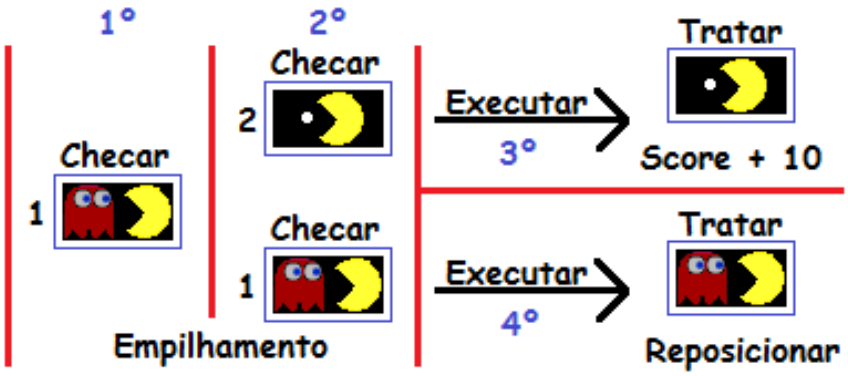


Figura 74 – Fluxo de execução com estratégia de escalonamento *DEPTH*

Neste sentido, inicialmente a primeira *Rule* (colisão entre *Ghost* e *Pacman*) seria escalonada e em seguida a segunda *Rule* (colisão entre *Pacman* e pastilha) seria igualmente escalonada. Ainda, o escalonador do tipo *DEPTH* faria com que a última *Rule* adicionada fosse a primeira a ser executada (*LIFO*), evitando o problema de ignorar o tratamento das colisões com as pastilhas.

Outrossim, a utilização do escalonador *BREATH* resultaria no mesmo problema encontrado na execução realizada sem o auxílio de um escalonador. Isso aconteceria porque esse escalonador particularmente realizaria a execução seguindo uma estrutura de dados do tipo fila, aprovando inicialmente a *Rule* que considera a colisão entre *Ghost* e *Pacman*, desconsiderando a outra *Rule* em questão. A Figura 75 ilustra o fluxo de execução do cenário exemplo considerando a estratégia de escalonamento do tipo *BREADTH*.

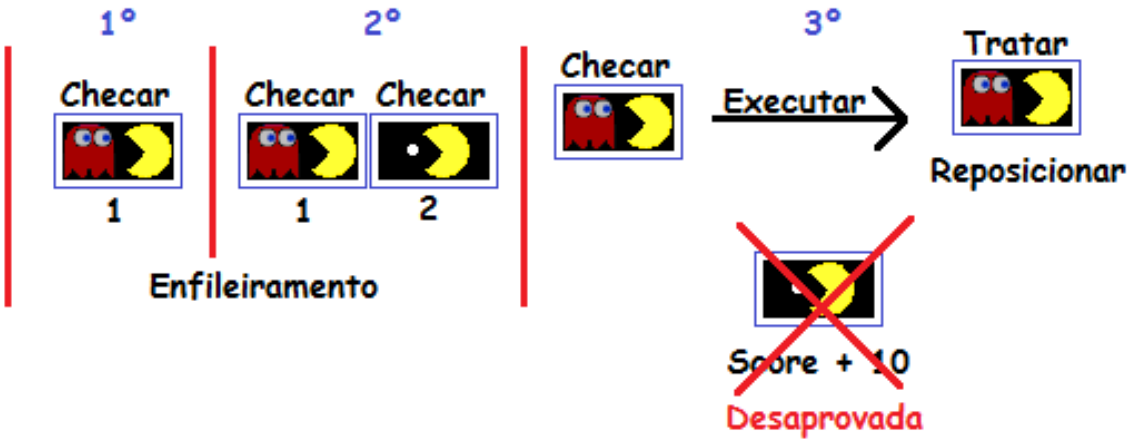


Figura 75 – Fluxo de execução com estratégia de escalonamento *BREADTH*

Apesar de o escalonador adequado ter resolvido o problema nesse caso em particular, esse tipo de questão dificultaria de certa forma a implementação de aplicações PON, uma vez que o desenvolvedor deveria estar ciente das implicações referentes à ordem com que as *Rules* são previamente definidas.

Em casos como esse, uma solução apropriada para minimizar as dificuldades oriundas de implementação cujas *Rules* impactem na execução de outras, seria a aplicação de níveis de prioridade na construção dessas. A correta aplicação de níveis de prioridade, bem como a utilização do escalonador de *Rules* do tipo *PRIORITY* evitaria o problema relatado.

Neste sentido, as *Rules* poderiam ser classificadas como de baixa prioridade (*low*), média prioridade (*medium*) e alta prioridade (*high*). Na solução do problema encontrado no Algoritmo 33, as *Rules* referentes à checagem de colisões com pastilhas deveriam ser classificadas com um nível de prioridade acima ao das *Rules* referentes à checagem de colisões com os fantasmas.

4.2 CASO DE ESTUDO – SISTEMA DE VENDAS – PON

As contribuições deste trabalho para com a composição de aplicações PON foram elucidadas em questões pontuais no âmbito dos casos de estudo apresentados na Seção 2.4. Basicamente, essas questões teórico-descritivas se deram em torno de dificuldades ou problemas encontrados no desenvolvimento de tais aplicações.

Diferentemente das seções anteriores, a seção corrente apresenta um comparativo quantitativo e, principalmente, qualitativo de uma nova implementação do sistema de pedido de vendas (previamente apresentado na Subseção 2.4.2). Tal comparativo busca analisar duas implementações distintas desse sistema no PON, contemplando os mesmos requisitos em ambas as versões. A primeira versão é desenvolvida com base nos recursos originais disponíveis no *framework*, enquanto a segunda versão é desenvolvida sob os princípios de alguns dos novos recursos, boas práticas e padrões propostos neste trabalho.

Em sua maioria esta seção foca em questões práticas com maior fundamentação experimental do que descritiva, uma vez que tais questões já foram tratadas pontualmente nas seções anteriores deste trabalho. Para isso, a Subseção 4.2.1 apresenta o escopo da aplicação, demonstrando e comparando a composição de *FBEs* e *Rules* nas duas versões. A Subseção 4.2.2, por sua vez, apresenta as otimizações pontuais vislumbradas para essa aplicação. A Subseção 4.2.3, particularmente, apresenta o fluxo de execução dessa aplicação. Por fim, a Subseção 4.2.4 apresenta as reflexões sobre a programação orientada a padrões.

4.2.1 Escopo da aplicação - Composição de *FBEs* e *Rules*

De maneira geral, o escopo dessa aplicação segue basicamente a descrição do caso de estudo apresentado na Subseção 2.4.2. Entretanto, sua regra de negócio apresenta algumas distinções na maneira com que a aplicação é executada. Em suma, as diferenças entre as implementações estão na maneira com que os descontos são concedidos a uma venda. Na versão original, existiam 20 tipos de descontos para serem aplicados no valor final da venda, de acordo com o tipo do cliente. A nova implementação, por sua vez, além desse desconto, considera descontos individuais e pontuais para cada item incluído na venda.

É importante observar, entretanto, que outras pequenas distinções, principalmente em relação à maneira com que as *Rules* são criadas, distinguem a implementação proposta neste presente trabalho com as apresentadas em [VENÂNCIO *et al*., 2011; SIMÃO *et al.,* 2012b], particularmente pelo fato das implementações apresentarem hibridismo com o POO. Nesse sentido, conforme

discutido anteriormente, este trabalho busca orientar os desenvolveres a conceber implementações no PON mais puras, o que é tido como uma dificuldade atualmente.

Para isso, a Figura 76 ilustra o diagrama de atividades que contempla a execução dos novos requisitos definidos para o sistema de pedido de vendas.

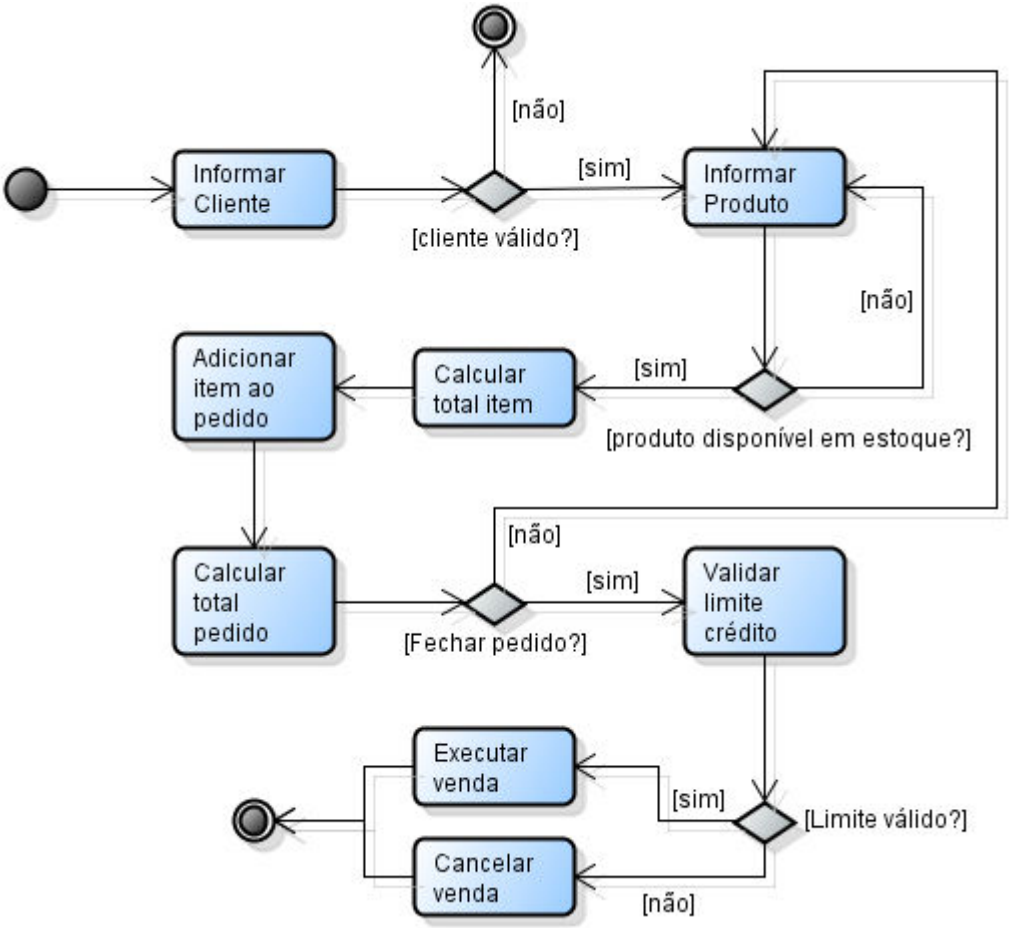


Figura 76 – Diagrama de Atividades - Sistema de pedidos de venda

Conforme apresenta o diagrama de atividades ilustrado na Figura 76, inicialmente o cliente é informado ao sistema. Uma vez confirmada a existência do cadastro do cliente, os itens de pedido (produtos) poderão ser informados. Para cada item de pedido inserido, com quantidade em estoque, o cálculo de seu valor total é realizado, baseado na quantidade, valor unitário e descontos. Assim, tal item é adicionado ao pedido que por consequência altera o valor total do pedido. Na sequência da execução, caso o cliente deseje inserir mais itens ao pedido, o fluxo retoma a adição de um novo produto e suas respectivas validações. No caso do pedido ser fechado, é realizada a validação do limite de crédito do cliente. Assim, caso o limite seja aprovado, a venda é realizada, caso contrário ela é cancelada.

Neste sentido, esta seção tem por objetivo apresentar as distinções entre a maneira como o sistema deveria ser implementado de acordo com os recursos disponíveis no *framework*. Para isso, a Subseção 4.2.1.1 apresenta a forma conhecida e usual de composição de *FBEs* e *Rules* apresentando sua implementação pré-padrões. A Subseção 4.2.1.2, por sua vez, demonstra o uso efetivo de algumas das novas funcionalidades propostas nesse trabalho, objetivando uma programação mais robusta pós-padrões.

4.2.1.1 Implementação pré-padrões

Inicialmente, a implementação dessa versão contempla todos os modelos (classes) para criação de *FBEs*. Tais modelos foram apresentados no diagrama de classes ilustrado na Figura 25. A criação desses modelos é essencial para a definição das entidades que atuam na execução do sistema. Sucintamente, a existência de entidades *Attribute* e *Method* em uma aplicação depende de um *FBE* que as comportem. Nesse sentido, o Algoritmo 35 apresenta um exemplo de classe que modela a criação de *FBEs Product*.

```
1  Product::Product(char* name, double price,
2                   int minimumStock, int currentStock)
3  {
4
5    atName = new String (this, name);
6    atPrice = new Double(this, price);
7    atMinimumStock = new Integer(this, minimumStock);
8    atCurrentStock = new Integer(this, currentStock);
9    atPercDescount = new Double(this, 0.0);
10   atTypeDiscount = new Integer(this, -1);
11
12 }
```

Algoritmo 35 – Modelo para criação de *FBEs Product*

Conforme apresentado no Algoritmo 35, o *FBE Product* apresenta em sua essência 6 *Attributes* distintos para posteriores vinculações com *Premises/Rules/Instigations*. Ainda, é possível observar que esse modelo apresenta parâmetros em seu construtor com o objetivo de inicializar alguns de seus *Attributes*. Ademais, nesse modelo em questão, nenhum *Method* foi definido.

Uma vez que os modelos para a criação de *FBEs* estejam implementados, a próxima etapa é instanciar os *FBEs* que farão parte da execução da aplicação. Para

isso, o trecho de código apresentado no Algoritmo 36 demonstra a inicialização da base de fatos.

```
1  void SalesOrderApp::initFactBase() {
2
3    double price = 100;
4    int minimumStock = 1;
5    int currentStock = 20;
6    int numProducts = 5;
7
8    productsList = new std::list<Product*>();
9
10   for (int i = 0; i < numProducts; i++) {
11     productsList->push_back(
12       new Product("Product", price, minimumStock, currentStock));
13   }
14
15   // . . . Demais instâncias de FBEs
16
17 }
```

Algoritmo 36 – Inicialização da base de fatos

Conforme apresentado no Algoritmo 36, a inicialização da base de fatos em questão cria uma lista de produtos (linha 8) para armazenar os produtos (*FBEs*) utilizados na aplicação. Nesse caso, foram criados precisamente 5 produtos com as mesmas características (preço, estoque mínimo e estoque corrente). Todos os demais *FBEs* referentes a esse caso de estudo são instanciados nesse método, entretanto foram omitidos nesse exemplo, conforme representa a linha 15.

Uma vez que estejam criados os *FBEs*, a próxima etapa de implementação de uma aplicação PON envolve a criação da base de regras. Nessa implementação em especial, as *Rules* são todas definidas no método *initRules*, conforme apresenta o Algoritmo 37.

```
1  void SalesOrderApp::initRules() {
2
3    // . . . Demais Rules da aplicação…
4
5    for (product = productsList->begin();
6         product != productsList->end(); ++product) {
7
8      prCheckInventory = new Premise(
9        (*product)->atCurrentStock, (*product)->atMinimumStock,
10       Premise::SMALLERTHAN, false);
11
12     rlRequestPurchase = new RuleObject(
13       "rlRequestPurchase", scheduler, Condition::SINGLE);
14     rlRequestPurchase->addPremise(prCheckInventory);
15     rlRequestPurchase->addInstigation(
16       new Instigation(department->mtRequestPurchase));
```

```
17
18      for (int i = 0; i < 20; i++) {
19        rlDiscountType = new RuleObject(
20          "rlDiscountType", scheduler, Condition::SINGLE);
21        rlDiscountType->addPremise(
22          (*product)->atTypeDiscount, i, Premise::EQUAL, false);
23        rlDiscountType->addInstigation(
24          (*product)->atPercDescount, i * 0.002);
25      }
26
27    }
28
29    // . . . Demais Rules da aplicação…
30
31  }
```

Algoritmo 37 – Composição de *Rules* para o controle de *FBEs Product*

O trecho de código descrito no Algoritmo 37 detalha a criação de *Rules* específicas para o controle dos *FBEs* produtos. As linhas 5 e 6 demonstram as iterações sobre a lista de produtos criadas anteriormente na inicialização da base de fatos. Ainda, para cada produto é criada uma *Rule* responsável por requisitar a compra de produtos (linhas 12 a 16) caso o estoque corrente esteja abaixo do estoque mínimo (linhas 8 a 10). Ademais, para cada produto são criadas 20 *Rules* distintas responsáveis pela concessão de descontos (linhas 18 a 25).

4.2.1.2 Implementação pós-padrões

Igualmente a versão/implementação anterior, o início do desenvolvimento na implementação orientada a padrões se dá pela composição dos modelos (classes) para criação de *FBEs*. Tais modelos, entretanto, poderiam seguir uma programação mais modular, onde cada módulo teria um dado conjunto de *Rules* e demais entidades colaboradoras. Tal abordagem proporcionaria maior dinamismo em relação a essa primeira abordagem considerada. Neste sentido, ao instanciar um *FBE*, não somente as entidades *Attribute* e *Method* seriam criadas, como também as demais entidades que compõem uma aplicação PON, tais como *Premises, Rules* e *Instigations*.

Ainda, a nova estrutura do *framework* viabiliza a composição de *Rules* ‘genéricas’ internamente aos modelos de *FBEs*. Desta forma, a cada instância de um *FBE* criado, todas as instâncias de *Rules* são devidamente criadas e vinculadas a esse *FBE* em questão. Essa abordagem facilita a composição de programas mais

modulares, uma vez que desmembra a base de fatos e de regras de uma estrutura centralizada (métodos iniciais de uma aplicação PON) e permite a criação/instanciação desses elementos em partes independentes do código.

Isso é viabilizado pelo compartilhamento de um único escalonador de *Rules* (*scheduler*) por meio do padrão *Singleton*, disponível assim em todo o escopo da aplicação. De modo a demonstrar um exemplo dessa implementação, o Algoritmo 38 apresenta o código da nova classe *Product* composta por suas respectivas *Rules*.

```
 1  Product::Product(string name, double price, int minimumStock,
 2    int currentStock, PurchasingDepartment *department)
 3  {
 4
 5    this->setName(name);
 6
 7    DOUBLE(this, atPrice, price);
 8    INTEGER(this, atMinimumStock, minimumStock);
 9    INTEGER (this, atCurrentStock, currentStock);
10    DOUBLE(this, atPercDescount, 0.0);
11    INTEGER(this, atTypeDiscount, -1);
12
13    Scheduler *scheduler = SingletonScheduler::getInstance();
14
15    RULE(rlRequestPurchase, scheduler, Condition::SINGLE);
16    rlRequestPurchase->addPremise(atCurrentStock, atMinimumStock,
17      Premise::SMALLERTHAN, Premise::STANDARD, false);
18    rlRequestPurchase->addMethod(department->mtRequestPurchase);
19
20    for (int i = 0; i < 20; i++) {
21      RULE(rlDiscountType, scheduler, Condition::SINGLE);
22      rlDiscountType->addPremise(atTypeDiscount, i,
23        Premise::EQUAL, Premise::STANDARD, false);
24      rlDiscountType->addMethod(this, atPercDescount, i * 0.002,
25        Attribute::STANDARD);
26    }
27
28  }
```

Algoritmo 38 – Modelo para criação de *FBEs Product* com *Rules* genéricas

Conforme apresenta o código ilustrado no Algoritmo 38, os *Attributes* são agora criados a partir da utilização dos pseudônimos (linhas 7 a 11). A linha 13 particularmente apresenta um ponteiro criado exclusivamente para comportar temporariamente a instância única do escalonador de *Rules*, necessário para a criação das *Rules* em tempo de execução. As *Rules* responsáveis pela concessão de descontos, bem como para requisição de compras, são implementadas agora no escopo do modelo do *FBE*. Sendo assim, o processo centralizado de criação de *Rules* (*initRules*) não precisa considerar tais *Rules*, se limitando à criação de eventuais *Rules* dependentes de muitos *FBEs*.

Ainda no Algoritmo 38 é possível observar uma peculiaridade nos parâmetros de criação de um *FBE Product*. No construtor de tal classe é exigida uma referência para outro *FBE* (departamento de compras), que somente faria uma associação com os produtos no sentido de viabilizar a criação de um vínculo com a requisição de compra de produtos, realizado por esse departamento.

Diferente da Programação Orientada a Objetos, onde um método faria a chamada de outros métodos a partir de referências (ponteiros) para objetos, na Programação Orientada a Notificações essa associação é realizada através de *Rules*, mantendo desacoplados todos os elementos que ditam o fluxo de execução da aplicação.

Ainda, de modo a demonstrar a vinculação de ambos os *FBEs,* o código exemplo ilustrado no Algoritmo 39 demonstra a criação de tais elementos seguindo a nova abordagem de implementação.

```
1  department = new PurchasingDepartment("Dep. Compras");
2
3  for (int i = 0; i < numProducts; i++) {
4    productsList->addElement(
5      new Product("Product", price, minimumStock,
6                  currentStock, department));
7  }
```

Algoritmo 39 – Exemplo de vinculação de dois *FBEs*

Conforme apresenta o Algoritmo 39, os *FBEs* seguem o estilo de criação tradicional, no âmbito do próprio método *initFactBase()*. Na verdade, a linha 1 representa a criação do *FBE* departamento de compras criado para posterior vinculação com cada novo *FBE* produto criado (linhas 5 e 6).

4.2.2 Otimizações pontuais

A flexibilidade proporcionada pela nova estrutura do *framework* possibilita a composição de programas mais eficientes em termos de desempenho. Neste âmbito, a presente subseção visa detalhar algumas das otimizações possíveis aplicadas pontualmente na implementação da aplicação em questão.

4.2.2.1 Estrutura de dados

A flexibilidade do padrão de projeto *Strategy* proporciona facilidades ao desenvolvedor para a utilização de diferentes estruturas de dados para uma mesma aplicação PON. Sendo assim, a estrutura de dados que comportaria as entidades PON pode ser alterada em tempo de execução, no momento de suas criações.

Neste âmbito, um cenário ideal é constatado na presença de *Attributes* com uma alta mudança de estados, vinculados a um número relativamente grande de *Premises.* Neste caso, a estrutura de dados denominada *NOPHASH* será sempre um forte candidato a ser utilizado para comportar esse tipo de entidade [VALENÇA, 2012].

Desta forma, em se tratando especificamente do estudo de caso do sistema de pedido de vendas, o *Attribute "atTypeDiscount"* referente aos descontos concedidos para os itens de um pedido, é um candidato a utilizar em seu escopo a implementação da estrutura de dados *NOPHASH.* O código ilustrado no Algoritmo 40 ilustra a implementação pontual de criação de apenas uma entidade composta por uma estrutura de dados do tipo *NOPHASH*.

```
1 CHANGE_STRUCTURE(SingletonFactory::NOPHASH);
2 INTEGER(this, atTypeDiscount, -1);
3 CHANGE_STRUCTURE(SingletonFactory::NOPVECTOR);
```

Algoritmo 40 – Exemplo de mudança de estrutura de dados

Na implementação do sistema de pedido de vendas, cada produto adicionado à cesta de compras de um determinado cliente apresentará 20 tipos distintos de descontos. Assim, quando adicionado a uma estrutura de dados com iteração linear (*e.g. NOPVECTOR*), todas as *Rules* seriam validadas e participariam do fluxo de notificações. Entretanto com a utilização da estrutura de dados *NOPHASH* somente o tipo de desconto específico será tratado. Assim, o cenário ideal para esse caso é a utilização pontual da estrutura *NOPHASH* (linha 1), em seguida é realizada a criação do *Attribute atTypeDiscount* (linha 2), o qual fará o uso da estrutura *NOPHASH* e finalmente retorna-se a utilização da estrutura de dados denominada *NOPVECTOR* (linha 3).

Nesse caso em particular, a mudança de estrutura de dados foi pertinente a apenas um *Attribute*, retornando à estrutura de dados do tipo *NOPVECTOR* (para a criação das demais entidades subsequentes) imediatamente após sua criação. É importante observar que caso o desenvolvedor ache pertinente, a estrutura de dados padrão não precisaria necessariamente ser retomada. Neste caso, todas as demais entidades a serem criadas nessa aplicação adotariam a última estrutura definida para a fábrica de entidades.

De maneira a analisar a eficiência e quantificar os benefícios do uso adequado e personalizado das estruturas de dados, um experimento foi realizado no sistema em questão. O experimento é representado pelo seguinte cenário:

- Número de clientes: 1
- Número de produtos: 100
- Quantidade no estoque: 10000
- Número de pedidos: 100
- Número de itens por pedido: 100 (cada um dos produtos)
- Quantidade de produtos por item de pedido: 10

A Figura 77 apresenta os resultados obtidos após a utilização de estruturas únicas (*i.e. NOPLIST*, *NOPVECTOR* e *NOPHASH*) em comparação com a utilização de estruturas mistas, no caso a utilização de *NOPVECTOR* e a utilização de *NOPHASH* para o caso pontual de concessão de descontos.

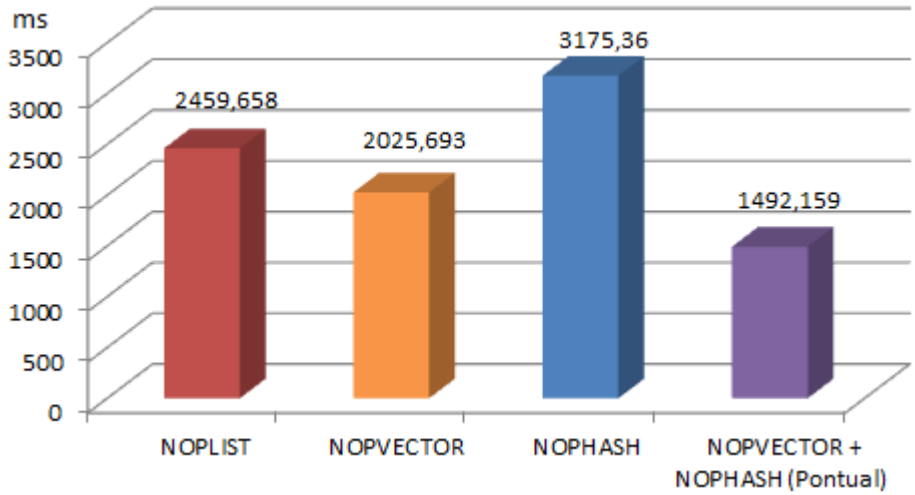


Figura 77 – Comparação entre implementações com estruturas únicas e mistas

Observa-se na Figura 77 certo ganho de desempenho com a utilização conjunta das estruturas de dados *NOPVECTOR* e *NOPHASH* em relação ao melhor caso alcançado por uma estrutura de dados única (*NOPVECTOR*). O ganho de desempenho nesse caso foi de cerca de 35,7%. Para o caso comparativo entre a estrutura *NOPHASH* e a implementação mista, o ganho foi de cerca de 112%.

É possível perceber inclusive que apesar de apresentar um caso em particular favorável à estrutura *NOPHASH* (*i.e.* concessão de descontos), tal estrutura é menos eficiente para as demais entidades desse sistema. Neste sentido, com a possibilidade de alterar a estrutura de dados em tempo de criação, criando entidades pontuais com a estrutura *NOPHASH* se mostra uma prática favorável e pertinente.

4.2.2.2 *Attributes* impertinentes

A presença de *Attributes* impertinentes nas aplicações não precisa necessariamente envolver casos onde esses apresentem muitas variações em seu estado. Existem casos que a avaliação de uma *Rule* é condicionada a um critério de parada ou depende de uma espécie de ‘ponta pé’ inicial.

No caso do sistema de pedido de vendas, mais especificamente, a cada inclusão de itens em um pedido, o valor total do pedido é atualizado baseado no preço, quantidade e desconto de cada item que o compõem. O valor total do pedido, por sua vez, faz parte de *Rules* que validam a aprovação/cancelamento da respectiva venda, comparando esse valor com o limite de crédito do cliente.

Neste âmbito, as *Premises* de tais *Rules* estariam constantemente sendo afetadas pelas variações no valor total do pedido, sendo que só precisariam verificar esta condição no momento em que a venda tiver sido fechada, ou seja, quando o cliente terminar de inserir/remover produtos na ‘cesta de compras’.

Os códigos que ilustram esse cenário são apresentados nos algoritmos Algoritmo 41, Algoritmo 42 e Algoritmo 43.

```
1  RULE(rlApproveProductItem, scheduler, Condition::SINGLE);
2  rlApproveProductItem->addPremise(
3    product->atCurrentStock, atQuantity,
4    Premise::GREATEROREQUAL, Premise::STANDARD, false);
5  rlApproveProductItem->addMethod(mtLowerStock);
6  rlApproveProductItem->addMethod(mtCalculateTotalValue);
7  rlApproveProductItem->addMethod(mtApproveSalesOrderItem);
8  rlApproveProductItem->end();
```

Algoritmo 41 – *Rule* responsável pela inclusão de itens em um pedido

O Algoritmo 41 apresenta a *Rule* responsável pelo controle de inclusão de itens em um pedido. A aprovação da *Rule* em questão depende apenas do produto escolhido possuir estoque suficiente para permitir sua inclusão no pedido (linhas 3 e 4). Assim, a execução dessa *Rule* reserva a quantidade de produtos informada, diminuindo-a do estoque (linha 5), bem como calcula o valor total desse item (linha 6), ou seja, seu valor unitário é multiplicado pela quantidade de produtos, podendo ainda apresentar eventuais descontos. Ademais, a aprovação desse item de pedido (linha 7) instiga a execução de outra *Rule*, cuja responsabilidade é manter o valor total do pedido atualizado. Tal *Rule* é apresentada no Algoritmo 42.

```
1  METHOD_OPERATION(this, mtCalculateTotalValue, atTotalSalesOrder,
2    new Addition(atTotalSalesOrder, salesOrderItem->atTotalValue),
3    Attribute::STANDARD);
4
5  RULE(rlApproveAndAddSalesOrderItem, scheduler, Condition::SINGLE);
6  rlApproveAndAddSalesOrderItem->addPremise(
7    salesOrderItem->atStatusSalesOrderItem, true,
8    Premise::EQUAL, Premise::STANDARD, false);
9  rlApproveAndAddSalesOrderItem->addMethod(mtCalculateTotalValue);
10 rlApproveAndAddSalesOrderItem->end();
```

Algoritmo 42 – *Rule* responsável pela manutenção do valor total de pedidos

Basicamente, a *Rule* ilustrada no Algoritmo 42 recalcula o total do pedido a cada item aprovado, com base no *Method* PON apresentado nas linhas 1 a 3. O mesmo princípio estaria presente no caso do cancelamento/remoção de um item desse pedido, no caso desse item apresentar estado falso.

O valor total do pedido, particularmente, influencia duas *Rules*, uma responsável pela aprovação e execução final de uma venda, enquanto a outra pelo cancelamento de tal venda. Entretanto, em um dado pedido poderiam ocorrer diversas inclusões e remoções de itens, os quais fariam com que o valor total desse pedido variasse a cada ação tomada pelo cliente. Nesse sentido, o valor total do

pedido seria impertinente a essas duas *Rules*, até que o pedido fosse dado como concluído pelo cliente. O código ilustrado no Algoritmo 43 apresenta esse cenário.

```
 1 PREMISE(prSalesOrderClosed, atCloseSalesOrder,
 2         true, Premise::EQUAL, Premise::STANDARD, false);
 3
 4 RULE(rlExecuteSalesOrder, scheduler, Condition::CONJUNCTION);
 5 rlExecuteSalesOrder->addPremise(prSalesOrderClosed);
 6 rlExecuteSalesOrder->addPremise(
 7   client->atStatus, Client::ACTIVE,
 8   Premise::EQUAL, Premise::STANDARD, false);
 9 rlExecuteSalesOrder->addPremise(
10   client->atCreditLimit, atTotalSalesOrder,
11   Premise::GREATEROREQUAL, Premise::IMPERTINENT, false);
12 rlExecuteSalesOrder->addMethod(client->mtExecuteSalesOrder);
13
14 RULE(rlCancelSalesOrder, scheduler, Condition::CONJUNCTION);
15 rlCancelSalesOrder->addPremise(prSalesOrderClosed);
16 rlCancelSalesOrder->addPremise(
17   client->atCreditLimit, atTotalSalesOrder,
18   Premise::SMALLERTHAN, Premise::IMPERTINENT, false);
19 rlCancelSalesOrder->addMethod(client->mtCancelSalesOrder);
```

Algoritmo 43 – *Rules* responsáveis pela aprovação/cancelamento de pedidos

Conforme apresentado no Algoritmo 43, tanto a *Premise* que confirma que o cliente possui limite de crédito suficiente para realizar tal compra (linhas 10 e 11), quanto a *Premise* contrária (linhas 17 e 18) são marcadas como passíveis de impertinência. Assim, a execução dessas fica condicionada ao 'ponta pé' originado pela confirmação do fechamento do pedido. Isso é representado pela *Premise* ilustrada nas linhas 1 e 2, sendo atribuídas as *Rule*s nas linhas 5 e 15, respectivamente.

De maneira a testar o impacto dessa implementação em comparação com a implementação padrão, o mesmo cenário comparativo ao utilizado na Subseção 4.2.2.1 é considerado. Neste teste, em particular, a melhor configuração foi selecionada (*i.e. NOPVECTOR* com a aplicação pontual de *NOPHASH*). Sendo assim, a Figura 78 ilustra os resultados de tal teste, alternando apenas a impertinência do *Attribute* para com as *Rules* responsáveis pela aprovação/cancelamento de pedidos.

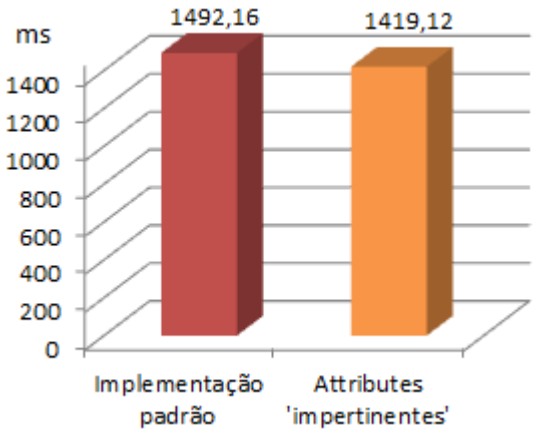


Figura 78 – Comparativo entre implementação padrão e *Attributes* 'impertinentes'

Conforme ilustra o gráfico da Figura 78, a eficiência proporcionada pela utilização de *Attributes* 'impertinentes' no cenário em questão representa aproximadamente a um ganho de desempenho de cerca de 5,1% em relação à implementação padrão. Nesse cenário em especial, tal recurso não representa um impacto muito grande no desempenho, pois a variação do estado do *Attribute* 'impertinente' é modesta. Entretanto, caso um sistema de maior escala seja considerado, onde muitos casos de impertinência estejam presentes, o impacto geral no desempenho seria extremamente pertinente.

4.2.3 Execução da aplicação

Na intenção de validar o fluxo de execução descrito nos parágrafos precedentes, bem como ilustrado no diagrama de atividades (Figura 76), a funcionalidade de *log* foi ativada na execução do cenário descrito e apresenta passo a passo o fluxo de notificações gerado pelo sistema, conforme demonstra a Tabela 4. Isso foi realizado de forma a garantir a execução de uma iteração completa de execução da aplicação em questão. Ainda o cenário permitiu a verificação e validação dos cálculos executados pela aplicação, como a soma de cada item de produto adicionado à cesta de compras do respectivo cliente.

Tabela 4 – Trecho do *log* gerado na execução do sistema de pedido de vendas

```
 1 = Product->atTypeDiscount = 2
 2 * rlDiscountType approved/executed
 3 = Product->atPercDescount = 0.004
 4 * rlApproveProductItem approved/executed
 5 = Product->atCurrentStock = 10
 6 = SaleOrderItem->atTotalValue = 996
 7 * SaleOrderItem->mtApproveSalesOrderItem executed
 8 = SaleOrderItem->atStatusSalesOrderItem = TRUE
 9 * rlApproveAndAddSalesOrderItem approved/executed
10 * SalesOrder->mtCalculateTotalValue executed
11 = SalesOrder->atTotalSalesOrder = 996
12 = Product->atTypeDiscount = 3
13 * rlDiscountType approved/executed
14 = Product->atPercDescount = 0.006
15 * rlApproveProductItem approved/executed
16 = Product->atCurrentStock = 0
17 * rlRequestPurchase approved/executed
18 = SaleOrderItem->atTotalValue = 994
19 * SaleOrderItem->mtApproveSalesOrderItem executed
20 = SaleOrderItem->atStatusSalesOrderItem = TRUE
21 * rlApproveAndAddSalesOrderItem approved/executed
22 * SalesOrder->mtCalculateTotalValue executed
23 = SalesOrder->atTotalSalesOrder = 1990
24 = SalesOrder->atCloseSalesOrder = TRUE
25 * rlExecuteSalesOrder approved/executed
26 Number of sales approved: 1
27 Number of sales canceled: 0
28 Number of purchase requests: 1
```

Conforme observado, a cada *Rule* aprovada e seu respectivo *Method* executado, geralmente se obtém uma variação de estado de um respectivo *Attribute,* o qual pode ser acompanho desde o início de sua criação. Assim todo o fluxo de execução fica extremamente claro sob o viés de verificação e validação da aplicação. Ademais, é possível verificar o determinismo de execução da aplicação PON desenvolvida, o que é importante, tendo em vista que seu fluxo de execução não é sequencial. Assim, caso necessário, o desenvolvedor PON poderá verificar o determinismo de execução conforme observado pela execução da funcionalidade *log* e adequar a ordem de criação de *Rules* ou ajustar a estratégia de escalonamento para garantir a correta execução de sua aplicação.

4.2.4 Reflexões

De maneira geral, a composição de modelos (classes) para criação de *FBEs* apresentou mudanças significativas em sua essência. Tais mudanças visaram

principalmente abrandar as dificuldades de programação presentes na nova estrutura do *framework*. Ademais, as contribuições relacionadas a otimizações na estrutura geral desse *framework*, bem como a flexibilidade de programação alcançada por meio das novas funcionalidades, possibilitaram a criação de aplicações PON mais eficientes. Por outro lado, tais contribuições aumentaram o conjunto de conhecimentos a serem compreendidos para a concepção de *software* nesse paradigma. Isso de certa forma aumenta a curva de aprendizado, o que de certo modo dificulta o entendimento e uso apropriado de todos os conceitos existentes para uma programação eficiente.

Neste sentido, o objetivo deste trabalho e principalmente dessa seção foi apresentar pontualmente exemplos práticos no âmbito da concepção de uma aplicação completa sob os princípios do PON. Buscou-se juntamente a essa abordagem apresentar as diferenças entre a programação tradicional e a programação orientada a padrões possibilitada na nova estrutura do *Framework* PON. A partir dessa visão norteada por exemplos, espera-se com que os desenvolvedores encontrem maiores facilidades para a criação de aplicações no PON.

Ainda, em relação à distribuição e organização de *Rules* em *FBEs*, proposta nessa seção, tal abordagem proporciona maior legibilidade e manutenibilidade ao sistema como um todo, uma vez que as *Rules* não ficam amontoadas em uma única classe. Ademais, apesar de a aplicação ilustrada apresentar um escopo moderado, o amontoamento de *Rules* em uma única classe dificulta o entendimento da aplicação como um todo, bem como dificulta a manutenção de tal aplicação, obrigando o desenvolvedor a conhecer todas as *Rules* que a compõem. Ainda, esse problema se agravaria em aplicações com escopos maiores tornando sua concepção ainda mais dificultosa.

Neste âmbito, em relação à ferramenta de *Log*, apesar de não ser mensurável sua importância quantitativamente, tal ferramenta representa uma poderosa forma de encontrar erros na execução de aplicações PON, especialmente em aplicações com um número de *Rules* considerável.

Diferente da programação imperativa, onde os elementos apresentam comportamentos passivos e, por esse motivo, não ditam o fluxo de execução de uma aplicação (*i.e.* dependem de um fluxo sequencial que os avalie), no PON mudanças em um *Attribute* influenciam todas as entidades interessadas em seu

estado, afetando imediatamente em seus comportamentos, o que representa a principal dificuldade no acompanhamento da execução do fluxo de execução de uma aplicação PON.

4.3 CONCLUSÃO

Em linhas gerais, esse capítulo apresentou a essência da programação no PON, com base em seu novo *framework*. O capítulo demonstrou de maneira linear os passos necessários para o desenvolvimento de uma aplicação nesse paradigma. Além disso, apresentou pontualmente a aplicação de padrões de desenvolvimento em casos particulares, aderindo uma visão prática, orientada a exemplos em casos de estudo reais. Essa abordagem contribuiu para um melhor entendimento da programação nesse paradigma como um todo.

Outrossim, em sua maioria, apesar deste capítulo ter apresentado algumas comparações quantitativas, o foco principal da proposição de padrões de desenvolvimento é simplificar o processo de desenvolvimento de aplicações. Neste sentido, o trabalho buscou procurar nortear os desenvolvedores quanto a decisões de implementação geralmente recorrentes, contribuindo para a minimização da curva de aprendizado desse paradigma, bem como contribuindo para o aceleramento do processo de desenvolvimento como um todo.

Quanto aos padrões de implementação, é importante ressaltar que apesar desse capítulo não reforçar o uso desses prolixamente, subentende-se que sua utilização é importante para a elaboração de uma codificação mais legível. Observa-se ademais que todos os algoritmos apresentados nas tabelas desse capítulo seguem tais padrões (detalhados na Seção 2.7), bem como adota os padrões de implementação específicos para o PON (Subseção 4.1.3). Além disso, a correta utilização dos pseudônimos representam contribuições nesse sentido.

5 CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo apresenta a conclusão final sobre o trabalho e aponta perspectivas para trabalhos futuros. Desta forma, a Seção 5.1 apresenta a conclusão dessa dissertação de mestrado relacionando às contribuições deste trabalho com os que antecederam suas origens. A Seção 5.2, por sua vez, apresenta vislumbramentos para trabalhos futuros que possivelmente contribuirão ainda mais para a maturação do estado da arte e da técnica do PON.

5.1 CONCLUSÃO

Este trabalho teve como principal objetivo elucidar a concepção de aplicações no PON. Para tanto, inicialmente tanto o PON quanto sua materialização na forma de *framework* foram detalhados sob o viés de padrões de projeto. Isso se deu sobre a descrição dos principais padrões de projetos elencados pela “Gangue dos Quatro” (*Gang of Four - GoF*) e utilizados para sua materialização. Neste âmbito, além de propiciar uma nova visão do PON e de sua materialização, o trabalho contribuiu com a reestruturação dessa materialização, identificando e refinando a aplicação dos padrões existentes na concepção de uma nova versão desse *framework* ou um novo *framework* melhor dizendo.

As questões relacionadas à implementação de padrões de projeto no âmbito do *Framework* PON ficam praticamente abstraídas ao desenvolvedor PON, atuando apenas em nível de execução. Por exemplo, o padrão *Iterator* é responsável por realizar a iteração entre as entidades PON; o padrão *Abstract Factory* é responsável pela criação de entidades PON; o padrão *Singleton* é responsável por centralizar a criação de entidades a partir de uma única instancia da fábrica PON; e o padrão *Command* é responsável pela delegação da execução do fluxo de execução de forma minimamente acoplada.

Há ainda principalmente o padrão *Observer,* o qual é considerado (por extrapolação) o âmago desse paradigma, responsável pela realização de notificações precisas e pontuais entre as entidades PON. Todos esses padrões

permitiram e viabilizaram a materialização dos conceitos do PON em forma de um *framework* ainda mais estruturado que o precedente.

Outrossim, alguns padrões, como o próprio padrão *Singleton* seria utilizado pelo desenvolvedor PON na concepção de aplicações. Particularmente, esse padrão propiciaria, por exemplo, uma melhor distribuição de criação de *Rules* em uma aplicação PON. Na verdade, esse padrão se mostrou importante ainda em outros casos, como na centralização da criação de entidades PON e na geração dos *logs* de execução. Em conjunto ao padrão *Singleton*, atua o padrão *Strategy* viabilizando a troca de contextos com apenas uma instrução.

Neste sentido, é possível afirmar que foi apropriado apresentar o PON sob o viés de padrões de projeto, uma vez que isso proporcionou um melhor detalhamento do funcionamento do paradigma e sua materialização. Além disso, a reestruturação do novo *framework* permitiu aumentar o nível de desacoplamento entre os elementos que compõem tal ferramenta, bem como flexibilizar e simplificar a programação como um todo. Neste âmbito, os desenvolvedores podem conceber código PON otimizado de acordo com as características de suas aplicações.

Em um segundo momento, o trabalho se apresentou como um estudo pioneiro, que visou principalmente nortear o desenvolvimento de aplicações no PON. Ainda, sob essa perspectiva, foram atribuídos novos conceitos e funcionalidades que proporcionaram maiores facilidades de composição de aplicações no PON, denominados Padrões de Desenvolvimento PON.

Tais padrões de desenvolvimento, de fato, agregaram no purismo do desenvolvimento no PON (operações aritméticas), bem como aproveitaram características sob o viés de desempenho (*Attributes* ‘impertinentes’, dependência de *Rules*). Cada conceito quando aplicado adequadamente garante qualidade no código desenvolvido e maiores facilidades para a manutenção e extensão das aplicações. É importante observar que as contribuições desse trabalho procuraram buscar um equilíbrio entre desempenho e flexibilidade, pendendo principalmente para o lado da flexibilidade.

Em suma, os padrões de desenvolvimento se mostraram adequados a um conjunto de necessidades de desenvolvimento. Ainda, foi possível observar que os detalhes de implementação influenciaram diretamente no fluxo de execução das aplicações e por consequência impactaram a execução em termos de desempenho.

Neste âmbito, é possível afirmar que os padrões quando bem aplicados se mostraram eficientes nos casos de estudo explorados.

Ainda, o trabalho apresentou um roteiro para a concepção de aplicações PON na materialização corrente, demonstrando pontualmente as particularidades na concepção dos elementos que fazem parte do processo de desenvolvimento de tais aplicações. Ademais, o trabalho apontou as principais dificuldades no âmbito de desenvolvimento de aplicações refletindo e apresentando possíveis soluções para essas dificuldades, tudo a luz dos padrões propostos.

Neste sentido, o trabalho foi preciso no quando e como utilizar determinados padrões, visando especialmente orientar o desenvolvimento de aplicações nesse paradigma. Tais orientações ocorreram em casos de estudo reais, o que de fato representa um facilitador para o entendimento da programação como um todo.

Ademais, é importante ressaltar que até então poucas aplicações haviam sido concebidas sob os princípios desse paradigma, o que de certa forma não contribuía para um aprendizado orientado a exemplos, oferecendo pouca instrução aos desenvolvedores PON. Além disso, os casos de estudo aqui apresentados buscaram atender diferentes domínios de aplicações, trazendo a tona certas dificuldades e possíveis soluções para essas. Neste âmbito, o trabalho deixa como legado, casos de estudo que enriquecerão a base de exemplos, proporcionando facilidades para a implementação de novas aplicações nesse paradigma.

Ainda no âmbito de desenvolvimento no PON, atualmente utiliza-se para execução desta tarefa, *IDEs* de desenvolvimento voltados a PI, especificamente para a Programação Orientada a Objetos (*e.g. Visual C++ Express Edition* e *Eclipse*). Apesar de tais *IDEs* possuírem excelentes depuradores de código, o mesmo não propicia certas facilidades de depuração de código PON, tendo em vista principalmente o fluxo não convencional das aplicações PON. Neste âmbito, foi concebida uma funcionalidade que facilita a depuração de código PON. Em verdade, tal funcionalidade garantiria ao desenvolvedor melhores alternativas sobre a verificação e validação de sua aplicação PON.

Outrossim, trabalhos paralelos [SIMÃO, 2005; BANASZEWSKI, 2009; SIMÃO *et al.,* 2012a] sugerem que aplicações desenvolvidas no PON seriam inerentemente eficientes, em virtude da redução de redundâncias temporais e estruturais viabilizada pela nova forma de conceber programas através de entidades reativas e notificantes. Entretanto, de nada adiantaria tais virtudes, se a concepção

de aplicações PON fosse realizada de maneira equivocada ou mesmo caótica. Nesse sentido, esse trabalho buscou padronizar a concepção de aplicações PON, propondo inclusive padrões de implementação, de modo a garantir certa organização e padronização no código. Isso agregaria em questões como maior legibilidade do código e consequentemente proporcionaria maior manutenibilidade e reusabilidade, entre outras características pertinentes e desejáveis.

É importante ressaltar os trabalhos realizados por [WIECHETECK *et al.*, 2011; WIECHETECK, 2011], no qual são estabelecidos os artefatos a nível de análise e projeto. Tais artefatos deveriam ser elaborados de antemão ao desenvolvimento de uma aplicação PON. Isso facilitaria a concepção de *FBEs* e suas *Rules* de modo a particularmente evitar inconsistências de execução da aplicação PON.

Ainda, uma primeira versão da elaboração da aplicação PON, poderia ser realizada através da ferramenta *Wizard* PON concebida nos trabalhos de [VALENÇA, 2012]. Deste modo, garantiria que uma primeira versão (código fonte) da aplicação PON fosse gerada automaticamente.

Neste âmbito, a união das contribuições aqui apresentadas, bem como as contribuições apresentadas nos demais trabalhos relacionados ao PON, abriria certamente margem para outros trabalhos que contribuiriam com o amadurecimento do paradigma em questão, o qual atualmente é considerado como um paradigma emergente.

5.2 TRABALHOS FUTUROS

O presente trabalho é pioneiro na apresentação dos conceitos do PON e de seu *framework* em forma de padrões de projeto, bem como no nortear da concepção de aplicações nesse paradigma pela proposta de Padrões de Desenvolvimento para o PON. A partir disso, esse trabalho abre perspectivas de pesquisa para maturar ainda mais a essência de composição de programas nesse paradigma. Neste sentido, a Subseção 5.2.1 abre margens para uma discussão importante, em relação à união da elaboração de artefatos de projeto, realizada a partir do DON, com a qualidade final da codificação gerada. A Subseção 5.2.2, por sua vez, vislumbra uma espécie de evolução ao *Framework* PON no sentido de prover maior inteligência às

entidades PON. A Subseção 5.2.3, particularmente, ressalta a necessidade de uma linguagem de programação, especialmente adequada ao uso de padrões, bem como a presença de um compilador PON. Por fim, a Subseção 5.2.4 reforça a necessidade de um ambiente multiprocessado e distribuído, sobretudo sobre a necessidade da existência de padrões de desenvolvimento para esse ambiente.

5.2.1 Evolução do *Wizard* PON

A elaboração de projetos a partir do DON e a possibilidade de gerar uma primeira versão do código-fonte através da ferramenta *Wizard* PON[8] possivelmente poderia levar tal ferramenta a uma solução para composição de *software* sob o viés de modelos, especificamente sob o viés de desenvolvimento dirigido a modelos, do inglês *Model Driven Architecture* (*MDA*).

Na verdade, a princípio, tal ferramenta já se apresenta apta para a elaboração de código a partir de interfaces gráficas. Neste sentido, bastaria a união de ambos os trabalhos para propor uma interface rica e intuitiva para a criação de modelos, de acordo com os princípios do DON.

Ainda, o código gerado por essa ferramenta precisaria se adequar aos padrões aqui propostos, onde tal código adotaria a organização e uma indentação única, bem como indicaria de certa forma aos desenvolvedores os padrões para a nomeação adequada de todas as entidades.

Conforme proposto no DON, em particular, é possível mapear a ideia de notificação para a habilitação de transição das Redes de Petri (RdP) [SIMÃO, 2005; BANASZEWSKI, 2009; WIECHETECK, 2011]. De forma a conceber aplicações em alto nível via modelos, uma ferramenta de RdP *open source* poderia ser estendida ao PON, tendo em vista suas naturezas de utilização. Desta forma, o desenvolvedor poderia conceber as entidades PON a partir de modelos elaborados em tais ferramentas.

[8] Sucintamente, a ferramenta *Wizard* PON representa um assistente para composição de *FBEs* e *Rules* em alto nível, através de uma interface amigável para o desenvolvedor. Tal ferramenta foi proposta recentemente nos trabalhos de [VALENÇA, 2012].

Ademais, as RdP poderiam ainda facilitar as melhorias relacionadas à depuração de código PON. Esta poderia estar atrelada a uma execução gráfica fornecida através da concepção de RdP, a qual permitiria ‘enxergar’ a execução de uma aplicação PON. Isso se daria a partir de um ambiente de desenvolvimento acoplado a esta ferramenta. Assim, isto seria possível, por exemplo, através de uma implementação de *plugins* para a *IDE* de desenvolvimento *Eclipse*.

Outrossim, a atual ferramenta *Wizard* PON apresenta certos problemas na composição de *Methods*, visto que a execução de uma aplicação PON é delegada a métodos do POO. Para isso, a ferramenta oferece um espaço para que o desenvolvedor insira código imperativo na construção de aplicações. Neste sentido, a proposta dos novos *Methods* PON para a composição de operações aritméticas abriria possibilidades para a elaboração de alguma modelagem para a composição de *Methods*.

Neste caso a implementação de *Methods* PON apresentaria maior purismo às aplicações, principalmente pelo evitar da intervenção de métodos da PI. Ademais, tal característica é extremamente importante em aplicações PON, tendo em vista a atual dificuldade de geração automática da dinâmica de execução de uma aplicação do POO por ferramentas do tipo *MDA*.

Ainda no âmbito de evoluções para com o *Wizard* PON, tal ferramenta também poderia ser dotada de mecanismos inteligentes que determinariam em tempo de montagem de uma aplicação as estruturas de dados que essas adotariam baseado em suas características e quantidade de notificações.

Na verdade, o mesmo princípio poderia ser aplicado a todos os padrões de desenvolvimento propostos nesse trabalho. Todo esse processo poderia ser automatizado e abstraído ao desenvolvedor, o que facilitaria enormemente a concepção de aplicações nesse paradigma, visto que os desenvolvedores só precisariam se preocupar com a lógica da aplicação.

5.2.2 Entidades PON mais inteligentes

A complexidade das aplicações PON tende a crescer à medida que o paradigma evoluir e suas ferramentas apresentarem maior maturidade. Tais aplicações apresentariam maior dinamismo, especialmente as que envolverem

interações com o usuário. Neste sentido, apesar de ferramentas como o *Wizard* PON poderem potencialmente determinar inteligentemente, em tempo de criação, as características dos elementos que compõem uma aplicação, como a estrutura de dados que cada entidade irá adotar, isso, em alguns casos, não seria previsível em tempo de montagem, ou poderia variar de entidade para entidade, tornando a montagem da aplicação ainda passível de previsões equivocadas.

Sendo assim, as entidades PON poderiam internamente aderir algum tipo de inteligência para se adaptarem à situações dinâmicas como essa. A utilização de *Attributes* 'impertinentes', por exemplo, poderia ser realizada automaticamente sob o viés da execução de uma aplicação PON. Deste modo, seria atribuída certa 'inteligência' ao próprio *framework* PON, o qual seria capaz de detectar os *Attributes* impertinentes a partir de uma avaliação de seu histórico de execução. A partir deste momento este Attribute seria categorizado internamente pelo *framework* como um *Attribute* do tipo impertinente.

Outrossim, cada *Premise* relacionada com um *Attribute* impertinente poderia ser considerada como *Premise* impertinente no desenvolvimento no PON, seja de maneira manual ou de maneira automática (*e.g.* poder-se-ia adicionar espertezas as *Premises*, *Conditions* e/ou *Rules* para detectar isso automaticamente via estratégias estatísticas). Neste sentido, cada *Condition-Rule* com *Premise* impertinente teria os mecanismos para evitar e desativar apropriadamente as notificações de *Attributes* impertinentes.

5.2.3 Linguagem de programação e compilador PON

Outro trabalho futuro relacionado a esse seria a necessidade de uma linguagem de programação adaptada a um compilador PON. Tal linguagem seria uma grande evolução em relação aos pseudônimos, que de certa forma já apresentam uma forma de definir a escrita dos códigos PON. É importante ressaltar que tal linguagem de programação poderia levar em consideração os padrões de implementação propostos nesse trabalho, de maneira a garantir certa padronização do código.

Ademais, para tornar os programas ainda mais eficientes quando desenvolvidos com o PON, se faz necessário a construção de um compilador

particular que otimize as relações entre os objetos participantes do mecanismo de notificações. Estas otimizações incluem a eliminação de estruturas de dados para armazenar os objetos notificados (*i.e. Premises* e *Conditions*), uma vez que os objetos notificantes e notificados seriam conectados em tempo de compilação.

Este compilador poderia consistir em uma extensão de compiladores de código aberto disponível para as linguagens de programação que viessem dar suporte ao PON. Esta extensão se encarregaria de compilar apenas as relações entre os objetos do mecanismo de notificações. Deste modo, os desenvolvedores continuariam fazendo uso da linguagem de programação pertinente e de todas as capacidades do compilador estendido [BANASZEWSKI, 2009].

Neste cenário, pode-se dizer que o ideal seria uma composição de entidades pertencentes à cadeia de notificação, determinadas em baixo nível. Neste sentido, o ideal seria conceber as relações entre os elementos participantes do cálculo lógico-causal, através da implementação direta via compilador. Assim, por exemplo, após a instanciação de uma *Rule*, por parte do desenvolvedor, o compilador interpretaria esta definição como uma palavra reservada e assim encadearia as relações entre os demais elementos participantes da cadeia de notificação, através da linguagem de máquina [BANASZEWSKI, 2009].

Ainda, este compilador poderia ser integrado à ferramenta *Wizard* PON e ser usado de maneira transparente ao programador. Com esta integração, a partir do desenvolvimento feito no *Wizard*, o compilador geraria o código de máquina ou *assembly* pertinente sem o programador precisar instigar os serviços deste compilador ou muito menos tomar conhecimento sobre o compilador que está sendo usado.

5.2.4 Ambiente Multiprocessado e Distribuído

Para que o PON seja adotado efetivamente na construção de aplicações multiprocessadas (paralelas e distribuídas) ainda se fazem necessárias algumas melhorias. Nestas melhorias há diferenças para ambientes que compartilham memória entre os nós de processamento (*i.e.* ambientes paralelos) e ambientes nos quais cada nó apresenta a sua memória particular (*i.e.* ambientes distribuídos) multiprocessadas (paralelas e distribuídas) ainda se fazem necessárias algumas

melhorias. Nestas melhorias há diferenças para ambientes que compartilham memória entre os nós de processamento (*i.e.* ambientes paralelos) e ambientes nos quais cada nó apresenta a sua memória particular (*i.e.* ambientes distribuídos) [BANASZEWSKI, 2009].

Em ambientes distribuídos, faz-se necessário conceber uma plataforma que automatize e torne transparente ao programador todas as questões de comunicação entre os nós de processamento remotos e um mecanismo de balanceamento de carga. A plataforma a ser concebida poderia adotar uma camada de software/middleware pré-definida como o *CORBA* (*Common Object Request Broker Architecture*), *DCOM* (*Distributed Component Object Model*) ou o *RMI* (*Remote Method Invocation*), estas auxiliariam na implementação das abstrações relativas à comunicabilidade entre os nós de processamento [BANASZEWSKI, 2009].

O mecanismo de balanceamento de carga poderia potencialmente ser concebido baseando-se, por exemplo, em soluções fundamentadas em algoritmos evolucionários. Assim, os objetos participantes do mecanismo de notificações poderiam ser distribuídos de forma balanceada e cooperar (por notificações) conforme os endereços definidos na plataforma [BANASZEWSKI, 2009].

Em ambientes paralelos, por sua vez, onde as particularidades de comunicação pela rede são desnecessárias, apenas a implementação do modelo de escalonamento de *Rules* seria suficiente para as aplicações executarem efetivamente. No entanto, em ambos os ambientes se faz necessário um modelo eficaz para a resolução de conflitos entre as *Rules*. Para isto, o modelo de resolução conflitos para CON desenvolvido por Simão (2005) deveriam ser utilizados/melhorados [BANASZEWSKI, 2009]. Neste sentido, há melhorias descritas nos pedidos de patentes [SIMÃO e STADZISZ, 2009b; SIMÃO *et al.*, 2010].

Ainda, a programação nesse ambiente apresentaria dificuldades principalmente no âmbito de controle de fluxo e determinismo, o que abriria certamente margens para a proposição de padrões de desenvolvimento específicos para essas questões. Ademais, outros padrões poderiam ser adotados na definição de entidades em relação à suas particularidades, tais como a maneira que executariam, onde executariam, de que forma se comunicariam etc.

REFERÊNCIAS BIBLIOGRÁFICAS

[AHMED, 1998] Suhail Ahmed. *CORBA Programming Unleashed*. Sams Pub., 1998.

[ALBERT, 1994] P. Albert. *ILOG Rules, Embedding Rules in C++: Results and Limits*. OOPSLA´94 -Workshop Embedded Object-Oriented Production Systems (EOOPS), 1994.

[BANASZEWSKI *et al.*, 2007] Roni Fábio Banaszewski, Jean Marcelo Simão, Cesar Augusto Tacla e Paulo Cézar Stadzisz. *Notification Oriented Paradigm (NOP) - A Software Development Approach based on Artificial Intelligence Concepts*. VI Congress of Logic Applied to Technology - LAPTEC 2007. Santos, 2007.

[BANASZEWSKI, 2009] Roni Fábio Banaszewski. *Paradigma Orientado a Notificações: Avanços e Comparações*. Dissertação de Mestrado, CPGEI/UTFPR, Curitiba, 2009.

[BANERJEE *et al.*, 1995] Prithviraj Banerjee, John A. Chandy, Manish Gupta, Eugene W. Hodges IV, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy e Ernesto Su. *The Paradigm Compiler for Distributed Memory Multicomputers*. IEEE Computer 28 (10), pp. 37-47, 1995.

[BARR, 2011] Michael Barr. *Embedded Systems Glossary*. Data de acesso: 18 de Setembro de 2011. Disponível em: http://www.netrino.com/Embedded-Systems/Glossary.

[BATISTA *et al.*, 2011] Márcio Venâncio Batista, Roni Fábio Banaszewski, Adriano Francisco Ronszcka, Glauber Zárate Valença, Robson Ribeiro Linhares, Paulo Cézar Stadzisz, Cesar Augusto Tacla e Jean Marcelo Simão. *Uma comparação entre o Paradigma Orientado a Notificações (PON) e o Paradigma Orientado a Objetos (POO) realizado por meio da implementação de um Sistema de Vendas*. COMTEL 2011. Lima, Peru, 2011.

[BECK, 2007] Kent Beck. *Implementation Patterns*. Addison-Wesley, 2007.

[BROOKSHEAR, 2006] Glenn Brookshear. *Computer Science: An Overview*. Addison Wesley, 2006.

[BUSCHMANN *et al.*, 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons, 1996.

[CHENG e CHEN, 2000] Albert Mo Kim Cheng e Jeng-Rung Chen. *Response Time Analysis of OPS5 Production Systems*. IEEE Transactions on Knowledge and Data Engineering, vol. 12, n.3, pp. 391-409, 2000.

[COULOURIS *et al.,* 2001] George Coulouris, Jean Dollimore, Tim Kindberg e Gordon Blair. *Distributed Systems - Concepts and Designs*. Reading, MA: Addison-Wesley, 2001.

[DEBIAN, 2012] Debian. *Download do Debian – Pacotes essênciais*. Disponível em: http://www.debian.org/distrib/netinst. Acessado em: 11 de Junho de 2012.

[DEEN, 2003] S. M. Deen. *Agent-Based Manufacturing: Advances in the Holonic Approach*. Springer, 2003, ISBN 3-540-44069-0.

[DEMARCO, 1979] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon Press Computing Series, 1979.

[DÍAZ *et al.*, 2007] Manuel Díaz, Daneil Garrido, Sergio Romero, Bartolomé Rubio, Enrique. Soler e José. M. Troya. *A component-based nuclear power plant simulator kernel: Research Articles*. Concurrency and Computation: Practice and Experience, 19 (5), pp. 593 - 607, 2007.

[FAISON, 2006] T. Faison. *Event-Based Programming: Taking Events to the Limit*. Apress, 2006.

[FORGY, 1982] Charles L. Forgy. *RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem*. Artificial Intelligence, vol. 19, pg 17-37 1982.
[FREEMAN *et al.*, 2004] Eric Freeman, Elisabeth Freeman, Bert Bates, Kathy Sierra e Elisabeth Robson. *Head First Design Patterns*. O'REILLY, 2004.

[FRIEDMAN-HILL, 2003] Ernest Friedman-Hill. *Jess in Action: Rule Based System in Java*. Greenwich, CT, USA: Manning Publications Co, 2003.

[GABBRIELLI e MARTINI, 2010] Maurizio Gabbrielli e Simone Martini. *Programming Languages: Principles and Paradigms. Series: Undergraduate Topics in Computer Science*. 1st Edition, XIX, 440 p., Softcover, 2010.

[GAMMA *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GARTNER, 2011] Gartner Inc. *Gartner Says PC Shipments to Slow to 3.8 Percent Growth in 2011; Units to Increase 10.9 Percent in 2012*. Disponível em: http://www.gartner.com/it/page.jsp?id=1786014. Data de acesso: 11 de Novembro de 2011.

[GAUDIOT e SOHN, 1990] J-L. Gaudiot e A. Sohn. *Data-Driven Parallel Production Systems*. IEEE Trans. On Software Eng.. V. 16. No 3, pg. 281-293, 1990.

[GIARRATANO e RILEY, 1993] Joseph Giarratano e Gary Riley. *Expert Systems: Principles and Practice*. Boston, MA: PWS Publishing, 1993.

[GRUVER, 2007] William A. Gruver. *Distributed Intelligence Systems: A new Paradigm for System Integration*. Proceedings of the IEEE Int. Conference on Information Reuse and Integration (IRI), pg 14-15, 2007.

[HUGHES e HUGHES, 2003] Cameron Hughes e Tracey Hughes. *Parallel and Distributed Programming Using C++*. Addison Wesley, 2003.

[JOHNSTON *et al.,* 2004] Wesley M. Johnston, J. R. Paul Hanna e Richard J. Millar. *Advances in Dataflow Programming Languages*. ACM Computing Surveys, Vol. 36, No. 1, March 2004, pp. 1–34. University of Ulster, 2004.

[KERNIGHAN e PLAUGHER, 1978] Brian W. Kernighan e P. J. Plaugher. *The Elements of Programming Style*. 2d. ed., McGraw-Hill, 1978.

[KAISLER, 2005] Stephen H. Kaisler. *Software Paradigm*. John Wiley & Sons, 2005.

[KANG e CHENG, 2004] Jeong A. Kang e Albert Mo Kim Cheng. *Shortening Matching Time in OPS5 Production Systems*. IEEE Trans. on Software Eng. V. 30, N. 7, 2004.

[KEYES, 2006] Robert W. Keyes. *The Technical Impact of Moore's Law*. IEEE solid state circuits society newsletter, IBM, T. J. Watson Research Center, 2006.

[KRASNER e POPE, 1988] Glenn E. Krasner e Stephen T. Pope. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*. Journal of Object-Oriented Programming, 1(3):26-49, August/September, 1988.

[KUMAR *et al.*, 2005] Vijay Kumar, Naomi Leonard e A. Stephen Morse. *Cooperative Control*. New York: Springer-Verlag, 2005.

[LAVENDER e SCHMIDT, 1996] R. Greg Lavender e Doglas C. Schmidt. *Active Object: An Object Behavioral Pattern for Concurrent Programming*. Em Pattern Languages of Program Design, Addison-Wesley, 1996.

[LEE e CHENG, 2002] Pou-Yung Lee e Albert Mo Kim Cheng. *HAL: A Faster Match Algorithm*. IEEE Trans. On Knowledge and Data Eng, vol. 14, no. 5, pg 1047-1058 2002.

[LIMA, 2008] Adilson da Silva Lima. *UML 2.0: do requisito à solução*. (3ª ed.). Editora Érica, 2008.

[LINHARES *et al.*, 2011] Robson Ribeiro Linhares, Adriano Francisco Ronszcka, Glauber Zárate Valença, Márcio Venâncio Batista, Fernando Augusto Witt, Carlos Raimundo Erig Lima, Jean Marcelo Simão e Paulo Cézar Stadzisz. *Comparações entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sob o contexto de um simulador de sistema telefônico*. COMTEL 2011. Lima, Peru, 2011.

[LISKOV, 1988] Barbara Liskov. *Data Abstraction and Hierarchy*. SIGPLAN Notices, 23,5, 1988.

[LOKE, 2006] S. Loke. *Context-Aware Pervasive Systems: Architectures for a New Breed of Applications*. 1st Edition, Auerbach Publications (Taylor & Francis Group – USA – 6000 Broken Sound Parkway NW Suite 300 Boca Raton, FL 33487-2742), December 7, 2006, ISBN-10: 0849372550, ISBN-13: 978-0849372551, 1. doi:10.1201/9781420013498, 2006.

[MANFREDINI *et al.,* 2002] Fabio Manfredini, Roberto Borelli, Marcos Antonio Quinaia e Jean Marcelo Simão. *Padronização da Arquitetura de um Meta-Modelo de Controle Holônico*. In: XIX Semana de Pesquisa e XIV Semana de Iniciação Científica da UNIOSTE, 2008, Guarapuava - PR. Anais do XIV Seminário de Pesquisa e XIV Semana de Iniciação Ciêntífica - UNICENTRO. Guarapuava: UNICENTRO, 2008. v. 01. p. 68-68.

[MARTIN, 2006a] Robert Cecil Martin. *SRP: The Single Responsibility Principle*. Data de acesso: 13 de Dezembro de 2011. Disponível em: http://www.objectmentor.com/resources/articles/srp.pdf.

[MARTIN, 2006b] Robert Cecil Martin. *The Open-Closed Principle*. Data de acesso: 13 de Dezembro de 2011. Disponível em: http://www.objectmentor.com/resources/articles/ocp.pdf.

[MARTIN, 2006c] Robert Cecil Martin. *The Liskov Substitution Principle*. Data de acesso: 13 de Dezembro de 2011. Disponível em: http://www.objectmentor.com/resources/articles/lsp.pdf.

[MARTIN, 2006d] Robert Cecil Martin. *The Interface Segregation Principle*. Data de acesso: 13 de Dezembro de 2011. Disponível em: http://www.objectmentor.com/resources/articles/isp.pdf.

[MARTIN, 2006e] Robert Cecil Martin. *The Dependency Inversion Principle*. Data de acesso: 13 de Dezembro de 2011. Disponível em: http://www.objectmentor.com/resources/articles/dip.pdf.

[MARTIN e MARTIN, 2006] Robert Cecil Martin e Micah Martin. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, 2006.

[MARTIN, 2008] Robert Cecil Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1ed. Prentice-Hall, 2008.

[MCLAUGHLIN *et al.*, 2006] Brett D. McLaughlin, Gary Pollice e Dave West. *Head First Object-Oriented Analysis and Design*. O'REILLY, 2006.

[MEYER, 1988] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[MIRANKER, 1987] Daniel P. Miranker. *TREAT: A better Match Algorithm for AI Production Systems*. Sixth National Conference on Artificial Intelligence - AAAI'87, pp. 42-47, 1987.

[MIRANKER *et al.*, 1990] Daniel P. Miranker, David A. Brant, Bernie Lofaso e David Gadbois. *On the Performance of Lazy Matching in Production Systems*. 8th National Conference on Artificial Intelligence AAAI (pp. 685-692). AAAI Press / The MIT Press, 1990.

[MIRANKER e LOFASO, 1991] Daniel P. Miranker e Bernie Lofaso. *The Organization and Performance of a TREAT-Based Production System Compiler*. IEEE Transactions on Knowledge and Data Engineering , III (1), pp. 3-10, 1991.

[MOORE, 1965] Gordon Earle Moore. *Cramming More Components Onto Integrated Circuits*. Electronics Magazine, 1965.

[NASA, 2012] NASA. *Beginner's Guide to Aeronautics – Parts of an Airplane*. Data de acesso: 21 de Abril de 2012. Disponível em: http://www.grc.nasa.gov/WWW/k-12/airplane/airplane.html

[OLIVEIRA e STEWART, 2006] Suely Oliveira e David E. Stewart. *Writing Scientific Software: A Guided to Good Style*. Cambridge University Press, 2006.

[PAES e HIRATA, 2008] C. E Barros Paes, C. M. Hirata. *RUP Extension for the Software Performance*. 32nd Annual IEEE International Computer Software and Applications (COMPSAC '08), pp 732-738, July 28 2008.

[PAN *et al.*, 1998] Juiyao Pan, Guilherme N. de Souza e Avinash C. Kak. *FuzzyShell: A large-scale expert system shell using fuzzy logic for uncertainty reasoning*. IEEE Trans. Fuzzy Syst., vol. 6, no. 4, pp. 563–581, Nov. 1998.

[PETERS, 2012] Eduardo Peters. *Coprocessador para Aceleração de Aplicações Desenvolvidas Utilizando Paradigma Orientado a Notificações (PON)*. Dissertação (Mestrado em Informática Industrial) - Programa de Pós Graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná, Curitiba, 2012.

[PIMENTEL e STADZISZ, 2006] Andrey Ricardo Pimentel e Paulo Cézar Stadzisz. *Application of the Independence Axiom on the Design of Object-Oriented Software Using the Axiomatic Design Theory.* Journal of Integrated Design & Process Science, v. 10, 2006.

[PITTMAN, 2011] Jamey Pittman. *The Pac-Man Dossier*. Disponível em: http://home.comcast.net/~jpittman2/pacman/pacmandossier.html. Acessado em: 16 de Junho de 2011.

[PRESSMAN, 2006] Roger S. Pressman. *Engenharia de* Software. 6ª ed. Editora Mc Graw Hill, 2006.

[RAYMOND, 2003] Eric Steven Raymond. *The Art of UNIX Programming*. Pp. 327, A. Wesley, 2003.

[REILLY e REILLY, 2002] David Reilly e Michael Reilly. *Java Network Programming and Distributed Computing*. Addison-Wesley, 2002.

[RONSZCKA *et al.*, 2011] Adriano Francisco Ronszcka, Danillo Leal Belmonte, Glauber Zárate Valença, Márcio Venâncio Batista, Robson Ribeiro Linhares, Cesar Augusto Tacla, Paulo Cézar Stadzisz e Jean Marcelo Simão. *Comparações quantitativas e qualitativas entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sobre um simulador de jogo*. COMTEL 2011. Lima, Peru, 2011.

[ROY e HARIDI, 2004] Peter Van Roy e Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[RUSSEL e NORVIG, 2003] Stuart J. Russell e Peter Norvig. *Inteligência Artificial*. ed. Editora Campus, 2003.

[SCHMIDT *et al.*, 2000] Douglas Schmidt, Michael Stal, Hans Rohnert e Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2: patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

[SCOTT, 2000] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., 2000.

[SEVILLA *et al.*, 2008] Diego Sevilla, José M. Garcia e Antonio Gómez. *Using AOP to Automatically Provide Distribution, Fault Tolerance, and Load Balancing to the CORBA-LC Component Model*. Parallel Computing: Architectures, Algorithms and Applications, C. Bischof, M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.), John von Neumann Institute for Computing, Jülich, NIC Series, Vol. 38, ISBN 978-3-9810843-4-4, pp. 347-354, 2007. Reprinted in: Advances in Parallel Computing, Volume 15, ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

[SHEN e JUANG, 2008] Victor R. L. Shen e Tony Tong-Ying Juang. *Verification of Knowledge-Based Systems Using Predicate/Transition Nets*. IEEE Transaction on Systems, Man, and Cybernetics - Part A: Systems & Humans, V.38, N.1, 2008.

[SIMÃO, 2001] Jean Marcelo Simão. Proposta de uma Arquitetura de Controle para Sistemas Flexíveis de Manufatura Baseada em Regras e Agentes. Dissertação de Mestrado, CPGEI/UTFPR, Curitiba, 2001.

[SIMÃO, 2005] Jean Marcelo Simão. *A Contribution to the Development of a HMS Simulation Tool and Proposition of a Meta-Model for Holonic Control*. Tese de Doutorado, CPGEI/UTFPR, 2005.

[SIMÃO e STASZISZ, 2002] Jean Marcelo Simão e Paulo Cézar Stadzisz. *An Agent-Oriented Inference Engine applied for Supervisory Control of Automated Manufacturing Systems*. In: J. Abe, & J. Silva Filho, Advances in Logic, Artificial Intelligence and Robotics (Vol. 85, pp. 234-241). Amsterdam, The Netherlands: IOS Press Books, 2002.

[SIMÃO e STASZISZ, 2008] Jean Marcelo Simão e Paulo Cézar Stadzisz. *Paradigma Orientado a Notificações (PON) – Uma Técnica de Composição e Execução de Software Orientado a Notificações*. Pedido de Patente submetida ao INPI/Brasil (Instituto Nacional de Propriedade Industrial) em 2008 e a Agência de Inovação/UTFPR em 2007. Nº INPI Provisório 015080004262. Nº INPI Efetivo PI0805518-1. Patente submetida ao INPI. Brasil, 2008.

[SIMÃO e STASZISZ, 2009a] Jean Marcelo Simão e Paulo Cézar Stadzisz. *Inference Process Based on Notifications: The Kernel of a Holonic Inference Meta-Model Applied to Control Issues*. IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans, Vol. 39, Issue 1, 238-250, Digital Object Identifier 10.1109/TSMCA.2008.20066371, 2009.

[SIMÃO e STASZISZ, 2009b] Jean Marcelo Simão e Paulo Cézar Stadzisz. *Mecanismo de Resolução de Conflito e Garantia de Determinismo para o Paradigma Orientado a Notificações (PON)*. Patent pending submitted to INPI/Brazil (Instituto Nacional de Propriedade Industrial) in 02/2010 and Innovation Agency of UTFPR in 2009.INPI Number: PI1000296-0., 2009.

[SIMÃO, STADZISZ e KÜNZLE, 2003] Jean Marcelo Simão, Paulo Cézar Stadzisz e Luis Allan Künzle. *Rule and Agent-oriented Architecture to Discrete Control Applied as Petri Net Player*. (G. Torres, J. Abe, M. Mucheroni, & C. P.E., Eds.) 4th Congress of Logic Applied to Technology - LAPTEC 2003 , 101, p. 217, 2003.

[SIMÃO, TACLA e STADZISZ, 2009] Jean Marcelo Simão, Cesar Augusto Tacla e Paulo Cézar Stadzisz. *Holonic Control Meta-Model*. IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans, 2009.

[SIMÃO *et al.*, 2002] Jean Marcelo Simão, Marcos Antonio Quinaia e Paulo Cézar Stadzisz. *Um Padrão Arquitetural para Sistemas Computacionais de Controle Supervisório*. In: II SugarLoafPLoP, 2002, Itaipava - RJ. II SugarloafPLoP, 2002.

[SIMÃO *et al.*, 2003] Jean Marcelo Simão, Paulo Cézar Stadzisz e Marcos Antonio Quinaia. *A Pattern System to Supervisory Control of Automated Manufacturing*

Systems. In: The Third Latin American Conference on Pattern Languages of Programming SugarLoafPLoP 2003, 2003, Porto de Galinhas - PE. SugarLoafPLOP Procedings 2003. Recife, PE: CIn / UFPE, 2003. p. 83-99.

[SIMÃO *et al.*, 2010] Jean Marcelo Simão, Cesar Augusto Tacla, Roni Fábio Banaszewski e Paulo Cézar Stadzisz. *Mecanismo de Inferência Otimizado do Paradigma Orientado a Notificações (PON) e Mecanismos de Resolução de Conflitos para Ambientes Monoprocessados e Multiprocessados Aplicados ao PON*. Patent pending submitted to INPI/Brazil (Instituto Nacional de Propriedade Industrial) in 03/2010 and Innovation Agency of UTFPR in 2010. INPI Number: PI1003736-5.

[SIMÃO *et al.*, 2012a] Jean Marcelo Simão, Roni Fábio Banaszewski, César Augusto Tacla e Paulo Cézar Stadzisz. *Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study*. Journal of Software Engineering and Applications (JSEA), 2012.

[SIMÃO *et al.*, 2012b] Jean Marcelo Simão, Danillo Leal Belmonte, Adriano Francisco Ronszcka, Robson Ribeiro Linhares, Glauber Zárate Valença, Roni Fábio Banaszewski, João Alberto Fabro, César Augusto Tacla, Paulo Cézar Stadzisz e Márcio Venâncio Batista. *Notification Oriented and Object Oriented Paradigm Comparison via Sale System*. Journal of Software Engineering and Applications (JSEA), 2012.

[SIMÃO *et al.*, 2012c] Jean Marcelo Simão, Danillo Leal Belmonte, Glauber Zárate Valença, Márcio Venâncio Batista, Robson Ribeiro Linhares, Roni Fábio Banaszewski, João Alberto Fabro, César Augusto Tacla, Paulo Cézar Stadzisz e Adriano Francisco Ronszcka. *A Game Comparative Study: Object-Oriented Paradigm and Notification-Oriented Paradigm*. Journal of Software Engineering and Applications (JSEA), 2012.

[SIMÃO *et al.*, 2012d] Jean Marcelo Simão, Paulo Cézar Stadzisz, Carlos Raimundo Erig Lima, Fernando Augusto Witt, Robson Ribeiro Linhares. *Paradigma Orientado a Notificações em Hardware Digital*. Pedido de Proteção Industrial e Pedido de Patente enviados à Agência de Inovação da UTFPR respectivamente em 11/05/2012 e 17/07/2012, Curitiba - PR, Brasil - Aguardando Aprovação da Agência para eventual envio para o INPI.

[SOMMERVILLE, 2004] Ian Sommerville, *Software Engineering*, 8th Ed. Addison-Wesley, 2004.

[SUTTER e ALEXANDRESCU, 2004] Herb Sutter e Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison Wesley, 2004.

[TANENBAUM e STEEN, 2002] Andrew S. Tanenbaum e Maarten van Steen. *Distributed Systems: Principles and Paradigms*, Prentice-HALL, 2002.

[THIBODEAU, 2011] Patrick Thibodeau. *O que custa mais: corrigir falhas em aplicações Java ou Cobol*. Data de acesso: 02 de Janeiro de 2012. Disponível em: http://computerworld.uol.com.br/gestao/2011/12/12/o-que-custa-mais-corrigir-falhas-em-aplicacoes-java-ou-cobol/.

[TIANFIELD, 2007] Huaglory Tianfield. *A New Framework of Holonic Self-organization for Multi-Agent Systems*. IEEE Int. Conf. on System, Man & Cyb., 2007.

[TILEVICH e SMARAGDAKIS, 2002] Eli Tilevich e Yannis Smaragdakis. *J-Orchestra: Automatic Java Application Partitioning*. 16th European Conf. on Object-Oriented Programming, pg 178-204, B. Magnusson (Ed), Springer, 2002.

[TUTTLE e EICK, 1992] Sharon M. Tuttle e Christoph F. Eick. *Suggesting Causes of Faults in Data-Driven Rule-Based Systems*. Proc. of the IEEE 4th International Conference on Tools with Artificial Intelligence, pg 413-416, Arlington, VA., 1992.

[VALENÇA, 2012] Glauber Zárate Valença. *Contribuição para a Materialização do Paradigma Orientado a Notificações (PON).* Dissertação de Mestrado, PPGCA/UTFPR, Curitiba, 2012.

[VALENÇA *et al.*, 2011] Glauber Zárate Valença, Roni Fábio Banaszewski, Adriano Francisco Ronszcka, Márcio Venâncio Batista, Robson Ribeiro Linhares, João Alberto Fabro, Paulo Cézar Stadzisz e Jean Marcelo Simão. *Framework PON, Avanços e Comparações*. Simpósio de Computação Aplicada (SCA 2011). Passo Fundo, Rio Grande do Sul, Brasil, 2011.

[WACHTER *et al.,* 2004] Bram De Wachter, Thierry Massart e Cédric Meuter. *dSL: An Environment with Automatic Code Distribution for Industrial Control Systems*. Proc. of the 7th Int. Conf. on Principles of Distributed Syst., 2003, La Martinique, France, V. 3144 of LNCS, pg 132-45, Springer, 2004.

[WATSON *et al.*, 2009] G. R Watson, C. E. Rasmussen e B. R. Tibbitts. *An integrated approach to improving the parallel application development process*. IEEE International Symposium on Parallel & Distributed Processing, pp 1 - 8, 2009.
[WATT, 2004] David A. Watt. *Programming Language Design Concepts*. J. Willey & Sons, 2004.

[WIECHETECK, 2011] Luciana Vilas Boas Wiecheteck. *Um Método para Projetos de Software usando o Paradigma Orientado a Notificações.* Dissertação de Mestrado, CPGEI/UTFPR, Curitiba, 2011.

[WIECHETECK *et al.*, 2011] Luciana Vilas Boas Wiecheteck, Paulo Cézar Stadzisz, e Jean Marcelo Simão. *Um Perfil UML para o Paradigma Orientado a Notificações (PON)*. COMTEL 2011. Lima, Peru, 2011.

[WOLF, 2007] Wayne Wolf. *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*. Morgan Kaufmann, 2007.

[ZECHNER, 2011] Mario Zechner. *Beginning Android Games*. Apress, 2011.